Anpassbarkeit von Software-Werkzeugen in prozessintegrierten Entwicklungsumgebungen

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Klaus Lambert Weidenhaupt

aus Immerath, jetzt Erkelenz

Berichter:

Universitätsprofessor Dr. rer. pol. Matthias Jarke
Universitätsprofessor Dr. rer. nat. Gregor Engels

Tag der mündlichen Prüfung: 11. Juli 2001

Kurzfassung

Komplexe Modellierungsaufgaben, wie sie in den frühen Phasen der Informationssystementwicklung, aber auch im ingenieurwissenschaftlichen Kontext auftreten, erfordern eine geeignete Unterstützung durch Software-Werkzeuge, die sich ohne großen Aufwand an organisations- und projektspezifische Methoden und Prozesse anpassen lässt. Konventionelle produktorientierte CASE-Umgebungen werden in dieser Hinsicht vielfach als zu inflexibel empfunden, da sie auf hartkodierten Annahmen über die zu unterstützenden Prozesse beruhen. Eine verbesserte Anpassbarkeit versprechen prozesszentrierte Entwicklungsumgebungen, die auf formalen, austauschbaren Prozessmodellen basieren. In existierenden Ansätzen addressiert die Prozessanleitung jedoch primär die administrative Ebene des Projektmanagements, während die Konsequenzen der Prozessmodellinterpretation für die am Arbeitsplatz zur eigentlichen Aufgabendurchführung verwendeten Entwicklungswerkzeuge bislang kaum betrachtet wurden. In solchen Systemen ist das Verhalten der Werkzeuge weitgehend von der modellgesteuerten Prozessanleitung entkoppelt.

In der vorliegenden Arbeit untersuchen wir, wie man von einer Prozess*zentrierung* zu einer Prozess*integration* von Entwicklungsumgebungen gelangt. Ausgehend von sechs zentralen Integrationsanforderungen entwickeln wir einen integrierten Modellierungsansatz, bei dem ein existierendes Rahmenmodell für kreative Prozesse um Konzepte zur Werkzeugmodellierung angereichert wird. Mithilfe anpassbarer Querbezüge zwischen Prozess- und Werkzeugmodell wird ein organisations- und projektspezifisches *Umgebungsmodell* konfiguriert, das die Auswirkungen von Prozessen auf das Werkzeugverhalten klärt und die Grundlage für feingranulare, interpretative Anpassbarkeit darstellt.

Umgesetzt wird der Ansatz in Form eines generischen, objektorientierten Implementierungs-Frameworks, bei dem große Teile einer prozessintegrierten Umgebung bereits als vorgefertigtes, leicht erweiterbares Softwaregerüst vorliegen. Neben der effizienten Neuentwicklung unterstützt das Framework auch die Einbindung existierender Werkzeuge, was wir am Beispiel von drei weit verbreiteten Fremdwerkzeugen demonstrieren. Insgesamt wird die Praktikabilität des Ansatzes durch die erfolgreiche Realisierung von vier prozessintegrierten Entwicklungsumgebungen aus den Anwendungsdomänen Requirements Engineering und chemische Prozessmodellierung validiert.

Abstract

Complex modelling tasks in the early phases of information systems development as well as in other engineering domains require suitable software tool support that can be easily adapted to organisation and project specific methods and processes. Conventional product-oriented CASE tools have often been criticized as being too inflexible as their behaviour is determined by hard-coded assumptions of the processes to be supported. So called process-centred engineering environments, which are based on explicit and exchangeable process models, allow greater adaptability but mainly address administrative processes at the project management level. The consequences of process model enactment on the interactive engineering tools used for the actual task performance have been studied much less. In those systems, tool behaviour is largely decoupled from process model enactment.

This thesis investigates the evolution from process-*centred* to process-*integrated* engineering environments. Starting with a set of six key integration requirements, we develop an integrated modelling approach, which extends an existing model for creative processes by additional tool modelling concepts. By establishing adaptable cross-relationships between the process and the tool model, a specific *environment model* is configured. The environment model defines the effects of the process models on the tool behaviour and lays the foundation for fine-grained, interpretative tool adaptability.

The approach has been realised in the form of a generic, object-oriented *implementation framework* that provides a set of prefabricated software components for the rapid development of a process-integrated environment. The framework also supports the integration of existing legacy tools which is demonstrated with three widespread commercial tools. The overall practicability of our approach has been validated through the implementation of four process-integrated environments in the application domains of requirements engineering and chemical process engineering.

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik V (Informationssysteme) der RWTH Aachen. Während dieser Zeit haben zahlreiche Menschen durch fachliche und persönliche Gespräche, Diskussion und Beiträge Einfluss auf meine Arbeit genommen. Mein Dank gilt insbesondere:

- Prof. Dr. Matthias Jarke, der mir die Durchführung dieser Arbeit ermöglichte und mich in allen Belangen leitete und unterstützte;
- Prof. Dr. Gregor Engels für das Interesse an der Arbeit und die Bereitschaft, das Koreferat zu übernehmen;
- meinen Kollegen aus der Arbeitsgruppe PRIME, Peter Haumer, Ralf Dömges, Stefan Zlatinsis, Jürgen Rack und Ralf Klamma, und insbesondere dem Leiter der Arbeitsgruppe, Klaus Pohl;
- den studentischen Hilfskräften und Diplomanden Sebastian Brandt, Markus Hoofe, Michael Mattern, Tobias Rötschke, Dirk Schmidt, Robert Weber;
- den Mitarbeitern und Kollegen in den Projekten CREWS und IMPROVE;
- meinen Eltern;
- Barbara.

Inhaltsverzeichnis

Teil 1 Einordnung der Arbeit 1

1 EINLEITUNG...3

1.1 Motivation und Problembeschreibung...3

1.2 Ziele und Aufbau der Arbeit...5

2 PROZESSORIENTIERTE UNTERSTÜTZUNGSFUNKTIONEN 9

2.1 Ein Klassifikationsmodell für Prozessunterstützungsfunktionen............9
2.1.1 Unterstützte Projektebene 11
2.1.1.1 Projektmanagement-Ebene...11
2.1.1.2 Arbeitsplatzebene 12
2.1.2 Integrationstiefe 17
2.1.3 Kontextbezogenheit 19
2.1.4 Anpassbarkeit...21
2.1.5 Unterstützungsmodi und Durchsetzungsgrad 23

2.2 Bewertung existierender Ansätze 25
2.2.1 Methoden- und Projekthandbücher..26
2.2.2 Hilfesysteme 27
2.2.3 Assistenten 30
2.2.4 Prozesszentrierte Umgebungen..33
2.2.4.1 Prozessmodellierung 33
2.2.4.2 Konzeptueller Aufbau prozesszentrierter Entwicklungsumgebungen 34
2.2.4.3 Schnittstellen einer PZEU 36

2.3 Fazit...40
2.3.1 Vergleich der Prozessunterstützungsansätze...................................41
2.3.2 Einordnung prozessintegrierter Werkzeuge.....................................42

3 INTEGRATIONSANSÄTZE 47

3.1 Perspektiven der Werkzeugintegration 47
3.1.1 Integration als Informationsmanagement...48
3.1.2 Integration als eine Menge von orthogonalen Dimensionen.............49
3.1.3 Integration im Spannungsfeld von Basismechanismen und Prozessen........49
3.1.4 Fazit 51

3.2 Integrationsvoraussetzungen 52

3.3 Integrationsanforderungen in prozessintegrierten Umgebungen..........55
3.3.1 Überblick...55
3.3.1.1 Datenintegration 56
3.3.1.2 Kontrollintegration 56
3.3.1.3 Präsentationsintegration 57
3.3.2 Datenintegration zwischen den Prozessdomänen 58

3.3.2.1 Motivation ... 58
3.3.2.2 Bewertung existierender Ansätze ... 60
3.3.2.3 Fazit ... 69
3.3.3 Prozessorientierte Mediation der Werkzeuginteraktionen ... 69
3.3.3.1 Motivation ... 69
3.3.3.2 Bewertung existierender Ansätze ... 72
3.3.3.3 Fazit ... 84
3.3.4 Beschreibung von Werkzeugdiensten ... 84
3.3.4.1 Motivation ... 84
3.3.4.2 Bewertung existierender Ansätze ... 86
3.3.4.3 Fazit ... 94
3.3.5 Synchronisation zwischen den Prozessdomänen ... 94
3.3.5.1 Motivation ... 94
3.3.5.2 Bewertung existierender Ansätze ... 97
3.3.5.3 Fazit ... 98
3.3.6 Prozesssensitive Benutzeroberflächen ... 99
3.3.6.1 Motivation ... 99
3.3.6.2 Bewertung existierender Ansätze ... 101
3.3.6.3 Fazit ... 102
3.3.7 Werkzeugunterstützter Aufruf von Prozessfragmenten ... 102
3.3.7.1 Motivation ... 102
3.3.7.2 Bewertung existierender Ansätze ... 103
3.3.7.3 Fazit ... 103

3.4 Fazit ... 104

Teil 2 Lösungskonzept ... 105

4 ÜBERBLICK ÜBER DEN LÖSUNGSANSATZ ... 107

5 INTEGRIERTE PROZESS- UND WERKZEUGMODELLE ... 109

5.1 Darstellung der Modelle ... 109
5.1.1 UML ... 110
5.1.2 O-Telos ... 110

5.2 Motivation für integrierte Prozess- und Werkzeugmodelle ... 111

5.3 Modellierung von Prozessfragmenten ... 113
5.3.1 Ziele und Anforderungen ... 113
5.3.2 PRIME-PM: das NATURE-Prozessmetamodell ... 114

5.4 Modellierung von Werkzeugen ... 117
5.4.1 Ziele und Anforderungen ... 117
5.4.2 PRIME-TM: Das Werkzeugmetamodell ... 118

5.5 Integration der Modelle ... 120
5.5.1 Ziel des Umgebungsmodells ... 120
5.5.2 PRIME-UM: Das Umgebungsmetamodell ... 121
5.5.2.1 Abbildung von Ausführungskontexten auf Werkzeugkategorien ... 121
5.5.2.2 Abbildung von Entscheidungskontexten auf Interaktionselemente.. 124

5.6 Beispiel für ein Umgebungsmodell 127

5.7 Fazit 129

6 INTEROPERABILITÄT VON PROZESSSPRACHEN 131

6.1 Motivation 131

6.2 Interoperabilität in prozessbasierten Systemen 133
6.2.1 Standards 134
6.2.1.1 WfMC-Referenzmodell 134
6.2.1.2 Austauschformate 135
6.2.2 Föderierte Interoperabilitätsansätze 138
6.2.2.1 ProcessWall-Ansatz 138
6.2.2.2 APEL 138
6.2.3 Prozesskomponenten 139
6.2.3.1 Open Process Components 139
6.2.3.2 Pynode 140
6.2.3.3 Rollenkooperation 140
6.2.4 Diskussion und Schlussfolgerungen 141

6.3 Komponentenorientierte Darstellung des NATURE-Prozessmodells .143
6.3.1 Prozesskomponenten 144
6.3.2 Schnittstellenmetamodell 147
6.3.3 Zusammenfassung 152

6.4 Schnittstellenbindung 152
6.4.1 M2-Modell 153
6.4.1.1 Prozesssprachen-M2-Modell 153
6.4.1.2 Bindungs-M2-Modell 154
6.4.2 Integrationsmethodik 157
6.4.2.1 Überblick 157
6.4.2.2 Beispiel: Integration von SLANG-Netzen 158

6.5 Fazit 162

Teil 3 Umsetzung und Anwendungserfahrungen 165

7 DAS PRIME-RAHMENWERK 167

7.1 Die PRIME-Gesamtarchitektur 168
7.1.1 Framework-basierter Entwurfsansatz 168
7.1.2 Überblick über die Gesamtarchitektur 170
7.1.2.1 Werkzeuge der Durchführungsdomäne 171
7.1.2.2 Prozessmaschine 172
7.1.2.3 Kommunikationsmanager 173
7.1.2.4 Prozessspuren-Server 173
7.1.2.5 Prozessbeobachter-Klienten 175
7.1.2.6 Administrations- und Metamodellierungswerkzeuge 177
7.1.2.7 Prozess-Repository 181

7.2 Interaktionsprotokoll zwischen den Prozessdomänen 182

7.2.1 Dynamische Sicht der Durchführungsdomäne 183
7.2.2 Dynamische Sicht der Leitdomäne... 188
7.2.3 Rolle des Kommunikationsmanagers 190
7.2.4 Zusammenfassung... 194

7.3 GARPIT: ein Framework für prozessintegrierte Werkzeuge............ 195
7.3.1 Anforderungen an das GARPIT-Framework 195
7.3.1.1 Funktionale Anforderungen... 195
7.3.1.2 Nichtfunktionale Anforderungen.. 196
7.3.2 Architektur 197
7.3.2.1 Darstellung 197
7.3.2.2 Teilsysteme in Überblick... 198
7.3.2.3 StateManager 202
7.3.2.4 MessageInterface 205
7.3.2.5 ContextManager 207
7.3.3 Implementierung 219
7.3.4 Beispielanwendung des GARPIT-Frameworks 220
7.3.4.1 Phase 1: Werkzeugmodellierung.. 220
7.3.4.2 Phase 2: Implementierung 221
7.3.4.3 Phase 3: Prozessmodellierung 224
7.3.5 Zusammenfassung... 226

7.4 Integration existierender Werkzeuge 226
7.4.1 Anforderungen an Werkzeugschnittstellen 227
7.4.2 Wrapper-Architektur 229
7.4.3 Validierung... 231
7.4.4 Zusammenfassung... 233

7.5 GARPEM: die generische Prozessmaschinenarchitektur 234
7.5.1 Grobstruktur des GARPEM-Frameworks 235
7.5.2 Beschreibung der wiederverwendbaren Klassen...................................... 236
7.5.2.1 Sprachliche Klassen... 236
7.5.2.2 Technische Klassen 238
7.5.2.3 Integration einer neuen Sprache 239
7.5.3 Kontrollmodell 240
7.5.3.1 Ausführungsmodell von Kontextkomponenten................................ 240
7.5.4 Verwandte Ansätze 244
7.5.5 Zusammenfassung... 245

7.6 Fazit 245

8 ANWENDUNGEN 247

8.1 Entwicklungshistorie.. 247

8.2 PRIME-IMPROVE 250
8.2.1 Überblick über SFB IMPROVE... 250
8.2.2 Werkzeuge der PRIME-IMPROVE-Umgebung 251
8.2.2.1 Stellung des Fließbilds im Gesamtprozess 251
8.2.2.2 Anforderungen an das Fließbildwerkzeug.. 252
8.2.2.3 Realisierung... 254
8.2.3 Positionierung von PRIME-IMPROVE im SFB-Prototypen................... 258

8.3 Beispielsitzung 259

8.3.1 Definition eines Prozessfragments....259
8.3.1.1 Ziel der Prozessunterstützung....259
8.3.1.2 Prozessmodellierung....260
8.3.1.3 Werkzeugmodellierung....264
8.3.2 Ausführung eines Prozessfragments....264
8.3.3 Zusammenfassung....269

8.4 Fazit....270

9 SCHLUSSBETRACHTUNGEN....271

9.1 Beiträge der Arbeit....271

9.2 Erfahrungen und kritische Bewertung....273
9.2.1 Sicht des Anwenders....273
9.2.2 Sicht des Methodeningenieurs....274
9.2.3 Sicht des Umgebungsentwicklers....274

9.3 Ausblick....275

Teil 1

Einordnung der Arbeit

Kapitel 1

Einleitung

1.1 Motivation und Problembeschreibung

Die Entwicklung technischer Systeme sieht sich mit wachsenden Herausforderungen konfrontiert. Sowohl bei der Software-Entwicklung, als auch in klassischen ingenieurwissenschaftlichen Disziplinen (Maschinenbau, Verfahrenstechnik, Elektrotechnik etc.) zwingt der mit der allgemeinen Globalisierung immens gestiegene Konkurrenzdruck die Unternehmen dazu, Produktentwicklungszyklen zu verkürzen, die Kosten von Produktentwicklung und Produktion zu verringern und die Qualität der Ergebnisse der Entwicklungsprozesse zu erhöhen. Gleichzeitig steigt die Komplexität der zu entwickelnden Systeme mit der Proliferation der technischen Möglichkeiten und der Technikdurchdringung immer größerer Lebensbereiche.

Allgemein hat sich die Erkenntnis durchsetzt, dass die Komplexität heutiger Systeme nur dann bewältigt werden kann, wenn Analyse- und Modellierungstätigkeiten, die der eigentlichen Systemkonstruktion vorangehen bzw. iterativ damit verzahnt werden, ein angemessener Stellenwert eingeräumt wird. Beispielsweise ziehen Fehler in der Anforderungsanalyse, etwa unvollständig erhobene oder falsch verstandene Kundenanforderungen, später aufwändige Änderungen nach sich und sind häufiger Grund für Kosten- und Zeitüberschreitungen.

Methoden

Um Analyse und Entwurf technischer Systeme effizient durchführen zu können, ist der Einsatz *systematischer Methoden* unabdingbar. Produktseitig geben Methoden dem Entwickler einen Grundvorrat an Modellierungskonzepten und Strukturierungskonstrukten für die Systembeschreibung an die Hand, die in einer passenden, oft grafischen Notation dargestellt werden. Aus Prozesssicht definieren Methoden mehr oder weniger strukturierte Vorgehensweisen für die zielgerichtete Anwendung der Modellierungskonstrukte.

Unterstützung durch rechnerbasierte Werkzeuge

Angesichts der Größe der entstehenden Systembeschreibungen gilt eine rechnerbasierte Werkzeugunterstützung für den praktischen Umgang mit Analyse- und Entwurfsmethoden mittlerweile als unverzichtbar. Seit Mitte der 80er Jahre entstanden eine Vielzahl akademischer und kommerzieller Entwurfsumgebungen, die dem Entwickler so genannte CASE-Werkzeuge[1] – oder allgemeiner: CAx-Werkzeuge – zur Erstellung, Verwaltung und Konsistenthaltung von Systemmodellen und –implementierungen an die Hand geben.

[1] CASE: Computer-Aided Software Engineering

CASE-Umgebungen haben die hohen Erwartungen in Bezug auf Produktivitätssteigerung und Qualitätsverbesserung häufig nicht erfüllen können und den seit Anfang der 90er Jahre immer wieder prognostizierten Durchbruch noch nicht ganz geschafft [Roth93; Iiva96]. Als kritischer Schlüsselfaktor für den Erfolg von CASE-Werkzeugen gilt deren Anpassbarkeit an organisations- und projektspezifische Bedürfnisse. Denert hebt beispielsweise hervor, dass „Methoden des Software Engineering .. keineswegs so reif [sind], daß man sie mit hohem Aufwand in Werkzeuge einbrennen sollte. Wir brauchen vielmehr eine Technik, die es nicht prohibitiv teuer macht, Methoden, Sprachen, Vorgehensweisen, Werkzeuge etc. weiterzuentwickeln, mit ihnen zu experimentieren, sie Unternehmen, Projekten, Anwendungen, Basissystemen und den Menschen anzupassen, die diese Projekte durchführen“ [Dene93, S. 164]. In der Praxis wird jedoch beklagt, dass CASE-Umgebungen zu inflexibel sind und nur wenig Spielraum für Anpassungen und Erweiterungen lassen [EmFi96; JaHu98]. Vielmehr orientieren sich die Hersteller von CASE-Umgebungen häufig eng an den Vorgaben populärer Methodenhandbücher (z.B. Rational Rose für UML [JaBR99]) und kodieren diese fest in ihre Werkzeuge.

Mangelnde Anpassbarkeit und Prozessunterstützung ist ein Hemmschuh für den erfolgreichen Einsatz von CASE-Umgebungen

Die Flexibilisierung von Entwurfsumgebungen wurde in den vergangenen 15 Jahren aus unterschiedlichen Perspektiven untersucht. Produktorientierte Anpassungsmechanismen sind Gegenstand der Forschung im Bereich Methodengestaltung (method engineering) [HoVe97; Harm97; Roll97]. So genannte MetaCASE-Umgebungen oder CASE-Shells (z.B. MetaEdit+ [KeLR95]; Kogge [EbSU97]; MetaView [SoTM88]) bieten Metamodell-basierte Interpretations- und Generierungsmechanismen, mit deren Hilfe Werkzeugumgebungen relativ einfach an geänderte oder neue Modellierungskonzepte und -notationen angepasst werden können. Hier liegen mittlerweile ausgereifte Lösungsansätze vor, so dass wir den produktorientierten Aspekt der Anpassbarkeit in dieser Arbeit nicht weiter vertiefen werden und uns auf *prozessbezogene* Aspekte konzentrieren.

Wechselwirkung zwischen Entwicklungsprozessen und Werkzeugen

Zwischen Prozessen und Entwicklungswerkzeugen existiert eine enge Wechselwirkung [JaBR99; JaJa95; Mont94; Mart98]. Zum einen ermöglichen Werkzeuge überhaupt erst Vorgehensweisen, die ohne sie gar nicht denkbar oder viel zu arbeitsintensiv und daher zu kostspielig wären, z.B. die durchgängige Dokumentation von Entwicklungstätigkeiten über den gesamten Projektlebenszyklus [Pohl99]. Umgekehrt gilt, dass der von einer Organisation verfolgte Entwicklungsprozess, ob er nun explizit spezifiziert ist oder nur implizit in den Köpfen der Mitarbeiter vorliegt, die Funktionalität der benötigten Werkzeuge determiniert. Jacobson et al. sprechen in diesem Zusammenhang von Prozessen als den „use cases of the tools“ [JaBR99]. Werkzeugunterstützung in einer Entwurfsumgebung muss also flexibel an sich weiter entwickelnde und geänderte Prozesse anpassbar sein. So fordern Jacobson et al.: „At every release of process there must also be a release of tools“.

Prozesszentrierte Entwicklungsumgebungen

Um Prozessaspekte in einer Entwurfsumgebung *sichtbar* und *anpassbar* zu machen, müssen die zu unterstützenden Prozesse zunächst geeignet konzeptualisiert werden. In diesem Zusammenhang hat sich in den letzten etwa 15 Jahren mit der (Software)-Prozessmodellierung [DeKW99; FiKN94; FuWo96; AmCF97] eines der aktivsten Teilgebiete innerhalb der Softwaretechnik etabliert. In so genannten *prozesszentrierten Entwicklungsumgebungen* bilden formale Prozessmodelle die Grundlage für die Steuerung des Entwicklungsprozesses. Der wesentliche Vorteil prozesszentrierter Ansätze liegt in der expliziten Definition der Vorgehensweisen und der damit verbundenen *leichteren Anpassbarkeit* an projektspezifische Bedürfnisse.

Während die formale Prozessmodellierung und die Prozessanleitung mittels rechnergestützter Modellinterpretation mittlerweile recht gut verstanden sind, wurden die Konsequenzen einer zunehmenden Prozessorientierung für die am Arbeitsplatz verwendeten Entwurfswerkzeuge bisher jedoch kaum betrachtet. Interaktive, funktional reichhaltige Werkzeuge werden nur grobgranular in Prozessmodellen berücksichtigt und mit der Prozessmodellinterpretation integriert. Aus der mangelnden Integrationstiefe resultieren schwerwiegende Probleme. Der Benutzer wird in seinen Werkzeugen mit für die aktuelle Aufgabe irrelevanten Funktionen konfrontiert und erfährt durch das Werkzeug keinerlei *aktive* Unterstützung bei der methodischen Aufgabendurchführung [JaHu98]. Aufgrund der inkrementellen Arbeitsweise mit interaktiven Werkzeugen steht die eigentliche Aufgabendurchführung außerhalb der Kontrolle der Prozessmodellausführung [Böhm98], oder – schlimmer noch – hartkodierte Prozessannahmen in den Werkzeugen konfligieren mit den Vorgaben aus dem Prozessmodell (Process-in-the-Tool-Syndrom [Mont94]). Eine prozessmodellkonforme Arbeitsweise kann so nicht gewährleistet werden und macht die Prozessmodellinterpretation zumindest auf der feingranularen Ebene des individuellen Entwicklerarbeitsplatzes wertlos.

Mangelnde Integrationstiefe in prozesszentrierten Umgebungen

1.2 Ziele und Aufbau der Arbeit

Gegenstand der vorliegenden Arbeit ist die Prozess*integration* von Entwicklungsumgebungen. Die dahinter stehende Fragestellung lautet:

> Mit welchen Modellen und Mechanismen kann die Arbeitsweise von und mit Softwarewerkzeugen einem definierten Arbeitsprozess untergeordnet werden und auf effiziente Art und Weise an sich ändernde Prozessdefinitionen angepasst werden?

Die systematische Ausarbeitung dieser Fragestellung und des vorgeschlagenen Lösungskonzepts findet in drei Schritten statt.

Problemanalyse: Vergleich und Bewertung existierender Ansätze

Das Ziel des ersten Teils der Arbeit besteht zunächst darin, die Stärken und Schwächen heutiger Prozessunterstützungsansätze zu analysieren und diese mit unserer Zielvorstellung einer prozessintegrierten Entwurfsumgebung zu kontrastieren (Kapitel 2). Zur Strukturierung der Analyse entwickeln wir ein Klassifikationsschema, das eine Bewertung hinsichtlich der Merkmale *unterstützte Projektebene*, *Integrationstiefe*, *Kontextbezogenheit*, *Anpassbarkeit* und Abdeckung des Spektrums unterschiedlicher *Unterstützungsmodi* ermöglicht. Der Vergleich ergibt, dass prozesszentrierte Umgebungen grundsätzlich die gerade in kreativen Entwurfsdomänen nötige Flexibilität hinsichtlich der Definition neuer Prozesse oder der Anpassung existierender Prozesse bieten, aber nicht die nötige Integrationstiefe mit der eigentlichen Werkzeugumgebung aufweisen.

Die Grundidee prozesszentrierter Umgebung, nämlich die Erfassung von Prozesswissen in expliziten Prozessmodellen, bildet also den Ausgangspunkt für unser Lösungskonzept. Darauf aufbauend liegt der Hauptbeitrag von Kapitel 3 in der Herleitung von sechs zentralen *Integrationsanforderungen*, die den Übergang von einer prozess*zentrierten* zu einer prozess*integrierten* Entwicklungsumgebung charakterisieren. In einem breiten Literaturüberblick diskutieren und bewerten wir zu jeder die-

ser Anforderungen existierende Integrationsansätze aus den Bereichen Prozessmodellierung, Datenintegration, Kommunikationsinfrastrukturen, komponentenbasierte Softwareentwicklung, Werkzeugspezifikation, Benutzeroberflächen und Softwareergonomie. Der Vergleich ergibt, dass zu Teilproblemen zwar Lösungsansätze vorliegen, die jedoch noch nicht zu einer integrierten Gesamtlösung zusammengeführt worden sind.

Lösungskonzept: Integrierte Prozess- und Werkzeugmodellierung

Kapitel 4 gibt einen kurzen Überblick über den von uns gewählten Lösungsansatz, der auf der Grundidee beruht, Wissen über Prozesse *und* Werkzeuge gleichberechtigt in konzeptuellen Modellen explizit zu erfassen und diese Modelle miteinander zu verzahnen.

Die modellierungsseitigen Grundlagen werden in Kapitel 5 geschaffen. Wir erweitern ein existierendes Rahmenmodell für die kontextbasierte Modellierung kreativer Prozesse um Konzepte zur feingranularen Werkzeugmodellierung. Mithilfe anpassbarer Querbezüge zwischen Prozess- und Werkzeugmodell wird ein organisations- und projektspezifisches *Umgebungsmodell* konfiguriert, das die Auswirkungen von Prozessen auf das Werkzeugverhalten klärt und die Grundlage für interpretative Anpassbarkeit darstellt.

Das von uns verwendete kontextbasierte Rahmenprozessmodell lässt Aspekte der präskriptiven Ablaufsteuerung noch bewusst offen. Kapitel 6 beschreibt einen Ansatz für die Einbettung existierender Ablaufformalismen in das Rahmenprozessmodell. Insbesondere ermöglicht der Ansatz die Interoperabilität kompositer, verschiedensprachlich definierter Prozessfragmente.

Umsetzung: Framework für prozessintegrierte Umgebungen

Die effiziente *Entwicklung* prozessintegrierter Umgebungen ist Gegenstand von Kapitel 7. Zur Umsetzung des integrierten Werkzeug- und Prozessmodellierungskonzepts haben wir einen *Framework-basierten* Entwurfsansatz gewählt, bei dem große Teile einer prozessintegrierten Umgebung in Form eines vorgefertigten, leicht erweiterbaren Softwaregerüstes vorliegen. Ziel dieser Vorgehensweise ist insbesondere, dass sich der Umgebungsentwickler nicht mehr um die schwierige architekturelle und technische Integration zwischen der Prozessmodellausführung und den Werkzeugen zu kümmern braucht, sondern nur noch das Framework an den dafür vorgesehenen Stellen um spezifische Werkzeugfunktionalität und Prozessmodelle anreichern muss. Besonderer Wert wird in der Darstellung auf softwaretechnische Aspekte der Wiederverwendbarkeit gelegt (Framework-Architektur, Variationspunkte, Zusammenspiel zwischen generischen und spezifischen Architekturkomponenten, Kontrollinversion).

Über die Neuentwicklung von Werkzeugen hinaus eignet sich das Framework auch zum Wrapping existierender Werkzeuge, sofern diese hinreichend offen sind. Wir leiten zunächst Anforderungen an die Laufzeitschnittstellen von Fremdwerkzeugen her und illustrieren dann die Möglichkeiten und Grenzen des Wrappingansatzes anhand der Integration von insgesamt drei weit verbreiteten, kommerziellen Werkzeugen.

Die Praktikabilität des vorgeschlagenen Lösungsansatzes wurde durch die Realisierung bzw. Einbindung von 17 Werkzeugen in insgesamt vier prozessintegrier-

ten Entwurfsumgebungen aus den Anwendungsbereichen Requirements Engineerung und Verfahrenstechnische Modellierung nachgewiesen. Kapitel 8 geht speziell auf die im Aachener Sonderforschungsbereich „Informatische Unterstützung übergreifender Prozesse in der Verfahrenstechnik" entstandene Umgebung PRIME-IMPROVE ein und illustriert die Anpassung und Nutzung einer solchen Umgebung anhand einer Beispielsitzung.

In der Schlussbetrachtung unterziehen wir unseren Ansatz einer kritischen Bewertung aus den Blickwinkeln des Anwendungsingenieurs, des Methodeningenieurs und des Umgebungsentwicklers und skizzieren offen gebliebene Fragen, die den Ausgangspunkt für zukünftige Forschungsarbeiten bilden können.

Zum Abschluss dieses einleitenden Überblicks wollen wir noch auf die Querbezüge der vorliegenden Arbeit zu mehreren anderen Arbeiten hinweisen, die am Lehrstuhl für Informatik V an der RWTH Aachen entstanden sind. In der Dissertation von Peter Haumer wird eine Umgebung zur multimedialen Szenarienanalyse im Requirements Engineering beschrieben [Haum00], die auf Grundlage des hier beschriebenen PRIME-Frameworks realisiert wurde. Ralf Dömges untersuchte in seiner Doktorarbeit den zur Prozessintegration und Werkzeuganpassung komplementären Aspekt der Nachvollziehbarkeit und Prozessdokumentation [Dömg99]. In der Habilitation von Klaus Pohl wurde ein konzeptuelles Rahmenwerk für die kontinuierliche Dokumentation von Requirements Engineering-Prozessen entwickelt, welches die in dieser und den vorgenannten Arbeiten entstandenen Ergebnisse zusammenführt [Pohl99].

Kapitel 2

Prozessorientierte Unterstützungsfunktionen

In diesem Kapitel nehmen wir eine Bestandsaufnahme existierender Ansätze zur Prozessunterstützung vor und kontrastieren diese mit unserer Zielvorstellung einer Entwurfsumgebung, in der *prozessintegrierte Werkzeuge* den Entwickler am technischen Arbeitsplatz feingranular, kontextsensitiv und adaptabel unterstützen.

Um unseren Ansatz der Prozessunterstützung durch prozessintegrierte Werkzeuge besser in das Gesamtspektrum möglicher prozessorientierter Assistenzfunktionen einordnen und bewerten zu können, entwickeln wir in Abschnitt 2.1 zunächst ein allgemeines Klassifikationsschema. Als wesentliche Charakteristika für prozessorientierte Unterstützungsfunktionen betrachten wir die unterstützte Projektebene (Abschnitt 2.1.1), die Integrationstiefe der Prozessunterstützung mit den übrigen Komponenten der Entwurfsumgebung (Abschnitt 2.1.2), die Kontextbezogenheit (Abschnitt 2.1.3) und Anpassbarkeit (Abschnitt 2.1.4) der Prozessunterstützung sowie den Durchsetzungsgrad in unterschiedlichen Unterstützungsmodi (Abschnitt 2.1.5)

Die konkreten Ansätze, die am Entwicklungsprozess beteiligten Entwickler mit Informationen über problembezogene Vorgehensweisen zu versorgen und unterstützend oder lenkend in den Entwurfsprozess einzugreifen, decken ein sehr breites Spektrum ab und werden in Abschnitt 2.2 eingehend behandelt. Sie reichen von klassischen Methoden- und Projekthandbüchern (Abschnitt 2.2.1) über online verfügbare Hilfesysteme (Abschnitt 2.2.2), Assistenten oder Interface-Agenten (Abschnitt 2.2.3) bis hin zu so genannten prozesszentrierten Entwurfsumgebungen (Abschnitt 2.2.4), die auf Basis explizit definierter Prozessmodelle und unter Bereitstellung zusätzlicher Anleitungswerkzeuge den Entwurfsprozess unterstützen.

Das zuvor entwickelte Klassifikationsschema erlaubt eine einfachere Gegenüberstellung der einzelnen Ansätze, die Identifikation von Schwachstellen heutiger Unterstützungssysteme sowie die Definition von Merkmalen einer „idealtypischen“ Prozessunterstützung durch prozessintegrierte Werkzeuge (Abschnitt 2.3).

2.1 Ein Klassifikationsmodell für Prozessunterstützungsfunktionen

Um die in Abschnitt 2.2 vorgestellten Ansätze zur Prozessunterstützung besser in ein Gesamtspektrum einordnen und den in dieser Arbeit vorgestellten Ansatz

davon abgrenzen zu können, stellen wir in diesem Abschnitt ein einfaches Klassifikationsschema vor. Zur Klassifikation der Unterstützungsansätze unterscheiden wir folgende fünf Kriterien:

Merkmale für Prozessunterstützungsfunktionen

- **Unterstützte Projektebene** (Abschnitt 2.1.1): Adressiert die Prozessunterstützung vornehmlich administrative Planungs- und Koordinationstätigkeiten des Projektmanagers oder wird die systematisch-methodische Durchführung der eigentlichen Entwurfsaufgaben durch den technischen Entwickler unterstützt?
- **Integrationstiefe** (Abschnitt 2.1.2): Wie eng ist die Prozessunterstützung mit den übrigen Komponenten der rechnerbasierten Entwicklungsumgebung, insbesondere den Entwicklungswerkzeugen, verschränkt?
- **Kontextbezogenheit** (Abschnitt 2.1.3): Wie präzise ist die Prozessunterstützung zum Zeitpunkt der Inanspruchnahme auf den aktuellen Arbeitskontext zugeschnitten?
- **Anpassbarkeit** (Abschnitt 2.1.4): Ist die Prozessunterstützung flexibel an organisations- und projektspezifische Bedürfnisse anpass- und erweiterbar und welcher Aufwand ist dafür erforderlich?
- **Unterstützungsmodi** (Abschnitt 2.1.5): Wie hoch ist der Durchsetzungsgrad der Prozessunterstützung, d.h. wie stark ist der Benutzer an die von der Prozessunterstützung gemachten Vorgaben gebunden?

Nicht betrachtete Klassifikationskriterien

Dieser Kriterienkatalog ist keineswegs vollständig und könnte noch um zusätzliche Facetten erweitert werden, die wir im Kontext dieser Arbeit jedoch aus Aufwandsgründen nicht erschöpfend behandeln können und daher bewusst ausblenden. Dazu zählt vor allem die Unterstützung der *Nachvollziehbarkeit* [GoFi94] abgelaufener Entwurfsprozesse als Grundlage für das Änderungsmanagement, die Wiederverwendung fallspezifischen Erfahrungswissens und die kontinuierliche Prozessverbesserung. Aspekte der Nachvollziehbarkeits-Unterstützung wurden schon in anderen Forschungsarbeiten im Umfeld des hier vorgestellten PRIME-Rahmenwerks eingehend behandelt; hier sei auf die bereits oben erwähnten Dissertationen von K. Pohl (Repository-Unterstützung für die Nachvollziehbarkeit von Requirements Engineering-Prozessen, [Pohl95]), R. Dömges (Anpassbare Nachvollziehbarkeitsstrategien [Dömg99]) und P. Haumer (Nachvollziehbarkeit zwischen konzeptuellen Modellen und multimedialen Szenarien [Haum00]) sowie die Habilitationsschrift von K. Pohl [Pohl99] verwiesen. Darüber hinaus beschränken wir uns in dieser Arbeit bewusst auf die Unterstützung *individueller Arbeitsprozesse* eines einzelnen Entwicklers. Die *Skalierbarkeit* eines Prozessunterstützungsansatzes in Richtung einer Kooperationsunterstützung für Gruppenprozesse mit möglicherweise mehreren hundert involvierten Entwicklern ist daher ebenfalls nicht Gegenstand unseres Klassifikationsmodells. Das gleiche gilt für die damit verwandte, unter dem Stichwort *Awareness* behandelte Fragestellung, wie man im laufenden Entwurfsprozess einzelne Mitarbeiter gezielt auf für sie relevante Ereignisse außerhalb ihres Arbeitsbereichs aufmerksam machen kann [DoBe92; MaFS97].

Im Gegensatz zu anderen Klassifikationsmodellen, z.B. dem Werkzeugintegrationsmodell von Wasserman [Wass90] oder dem Requirements Engineering-Modell von Pohl [Pohl94], sprechen wir bei den einzelnen Kriterien unseres Klassifikationsmodells nicht von Dimensionen, da die Kriterien teilweise nicht völlig orthogonal zueinander sind. Das bedeutet, dass die Klassifikation eines Unterstüt-

zungsansatzes hinsichtlich eines Kriteriums unter Umständen seine Einordnung innerhalb eines anderen Kriteriums beeinflusst und dass nicht jede Kombination von Einordnungen unter die jeweiligen Kriterien sinnvoll ist. Zum Beispiel setzt die Kontextsensitivität eines Prozessunterstützungsansatzes einen gewissen Grad der Integration mit den Werkzeugen der Entwicklungsumgebung voraus. Nichtsdestotrotz charakterisiert jedes Kriterium einen spezifischen Aspekt, der nicht bereits vollständig von einem oder mehreren anderen Kriterien abgedeckt wird.

2.1.1 Unterstützte Projektebene

Die in einem Entwicklungsprojekt ablaufenden Tätigkeiten lassen sich grob in administrative Aktivitäten auf der *Projektmanagementebene* und technische Entwicklungstätigkeiten auf der *Arbeitsplatzebene* unterscheiden [NaWe99]. Während die eigentliche Durchführung der Entwicklung auf der Arbeitsplatzebene stattfindet, übernimmt die Projektmanagementebene Planungs-, Verwaltungs- und Kontrollaufgaben (siehe Abb. 1). Ansätze zur Prozessunterstützung lassen sich danach differenzieren, welche der beiden Ebenen sie vornehmlich adressieren. Im Folgenden skizzieren wir grob die auf den beiden Projektebenen anfallenden Aufgabenbereiche und charakterisieren so die aus Sicht der jeweiligen Projektebene benötigten Prozessunterstützungsfunktionen.

2.1.1.1 Projektmanagement-Ebene

Prozessunterstützung für das *Projektmanagement* befasst sich mit der Projektplanung und -organisation, der Ressourcenverwaltung und -koordination sowie der Anleitung und Kontrolle von Entwicklungsaktivitäten ([ThTh97], siehe oberen Teil von Abb. 1).

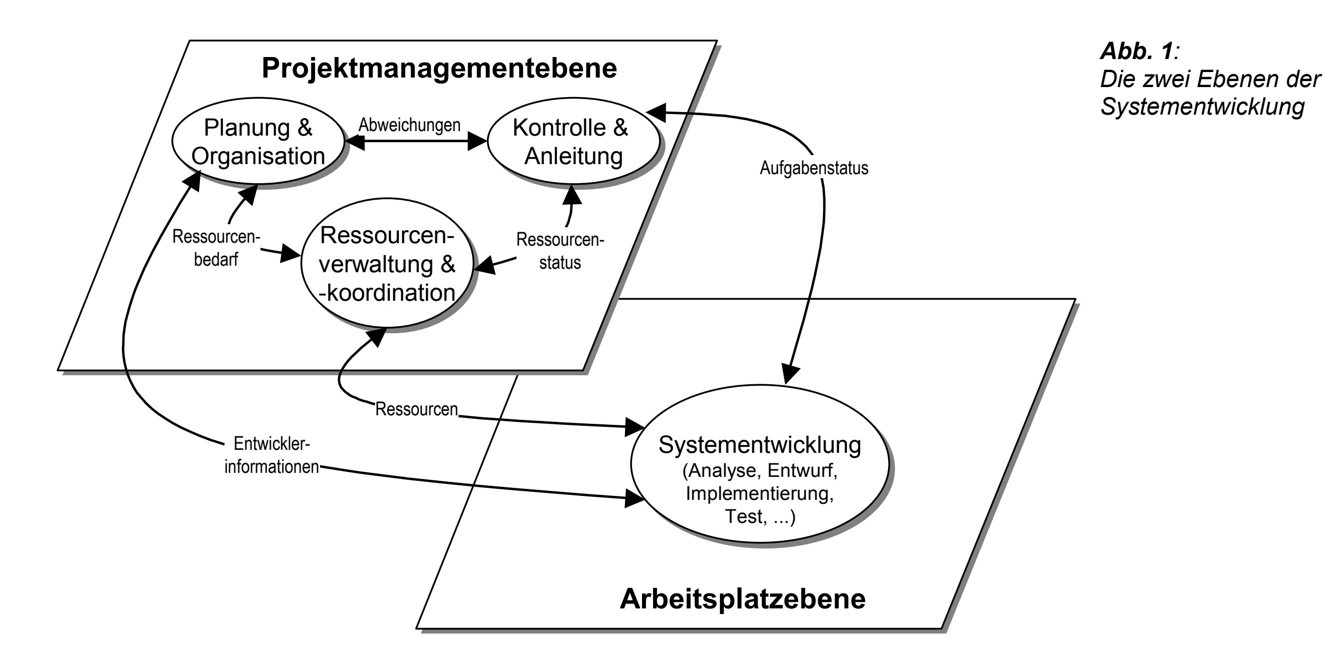


***Abb. 1**: Die zwei Ebenen der Systementwicklung*

Projektplanung und -organisation

Zu den Aufgaben der *Projektplanung und -organisation* gehört die Festlegung eines projektweiten, arbeitsplatzübergreifenden Entwicklungsprozesses in einem Projektplan und dessen kontinuierliche Fortschreibung und eventuelle Anpassung im

Laufe des Projektfortgangs. In einem Projektplan werden das Projektziel, d.h. die erwarteten und vertraglich festgelegten Endergebnisse, sowie die Einschränkungen und Randbedingungen, denen das Projekt unterliegt (Projektdauer, Budgetvorgaben, Verfügbarkeit von Ressourcen), festgehalten. Wichtigstes Hilfsmittel der Projektplanung ist die Dekomposition des Gesamtvorhabens in abgrenzbare Phasen und die weitergehende Strukturierung des Entwicklungsprozesses in eine Abfolge einzeln handhabbarer Aufgaben. Jede Aufgabe wird durch Angabe der für die Bearbeitung benötigten Ressourcen (Eingabedokumente), der erwarteten Ergebnisse (Ausgabedokumente) und in der Regel auch durch eine allgemeine Aufgabenbeschreibung näher charakterisiert. Explizit festgelegte Abhängigkeiten zwischen einzelnen Aufgaben (z.B. Datenflussabhängigkeiten, wenn eine Aufgabe die Ergebnisse einer anderen Aufgabe als Eingabedokument benötigt) definieren eine partielle Ordnung zwischen den Aufgaben und bilden die Grundlage für einen Terminplan, in dem innerhalb der festgesetzten Projektdauer Aufgabenanfang und -ende festgelegt werden. Die Definition von Meilensteilen dient der Beurteilung des Projektfortschritts während der Projektdurchführung. Da im Projektmanagement primär administrative Ziele verfolgt werden, wird die Aufgabenstruktur im Allgemeinen nur so weit verfeinert, wie es für die arbeitsteilige Zuordnung von Aufgaben an verschiedene technische Entwickler und deren Koordination erforderlich und sinnvoll ist. Die Internstruktur einzelner Aufgaben wird innerhalb des Projektmanagements nicht weiter betrachtet.

Ressourcen-verwaltung und -koordination

Im Rahmen der *Ressourcenverwaltung und -koordination* ist das Projektmanagement zum einen für die Festlegung einer Organisationsstruktur verantwortlich, die die Verantwortlichkeiten und Qualifikationen des im Projekt zur Verfügung stehenden Personals beschreibt und so eine Zuordnung von Aufgaben zu Prozessausführenden erlaubt. Zum anderen fällt die Beschaffung technischer Ressourcen und die Verwaltung und Koordination der erstellten Dokumente beziehungsweise deren Versionen und Konfigurationen in diesen Bereich.

Projektdurchführung

Im Laufe der *Projektdurchführung* wird schließlich auf Basis des Projektplans und unter Verwendung der zur Verfügung stehenden Ressourcen das Projektpersonal angeleitet und motiviert sowie der Projektfortschritt und Ressourcenverbrauch kontrolliert (vgl. Aufgabenbereich *Anleitung und Kontrolle* in Abb. 1). Die Entwickler am Arbeitsplatz werden über die aktuell anstehenden Aufgaben informiert und mit den dafür erforderlichen Arbeitsanweisungen und Ressourcen versorgt. Die zeit- und kostengerechte Erfüllung der Aufgaben wird anhand der vorher definierten Meilensteine überprüft. Bei Abweichungen werden gegebenenfalls korrigierende Maßnahmen veranlasst (z.B. Anpassungen des Terminplans, Budgeterhöhungen, personelle Veränderungen oder gar die Redefinition der Projektziele). Auf diese Weise wird das Zusammenspiel von Entwicklungsaufgaben und Entwicklern koordiniert und kontrolliert.

2.1.1.2 Arbeitsplatzebene

Durchführung von Aufgaben

Prozessunterstützung auf der Arbeitsplatzebene befasst sich mit der *methodischen Anleitung* des einzelnen Entwicklers bei der *systematischen Durchführung* von Aufgaben, die ihm vom Projektmanagement zugewiesen wurden. Die in einem typischen (Software-)Entwicklungsprozess anfallenden technischen Aufgaben lassen sich in unterschiedliche Arbeitsbereiche einordnen; u.a. werden Anforderungsanalyse, Systementwurf, Implementierung, Test, Installation, Dokumentation, Wartung und

Qualitätssicherung unterschieden [Nagl90]. Wie bereits in Kapitel 1 motiviert wurde, konzentrieren wir uns im Kontext dieser Arbeit auf Prozessunterstützung für die frühen Phasen der Systementwicklung, d.h. für die Anforderungsanalyse und den Entwurf. In diesen Phasen entfällt ein wesentlicher Teil der Entwicklungsaktivitäten auf Modellierungsaufgaben, in denen das neu zu entwickelnde oder auch schon bestehende System auf unterschiedlichen Abstraktionsebenen und aus unterschiedlichen Perspektiven analysiert und mit Hilfe geeigneter Modellierungstechniken beschrieben wird.

***Tab. 1**: Überblick über die gängigsten Methoden*

Methode	**Typ**	**Modellierungstechniken**	**Publikationen**
ARIS	Geschäftsprozess-modellierung	Erweitertes Entity-Relationship Modell,Funktionsbaum, Organigramm, Erweiterte ereignisgesteuerte Prozessketten, Wertschöpfungskettendiagramm	[Sche98]
BSP, Business Systems Planning	Geschäftsprozess-modellierung	Problem Table, Process/Entity Matrix, Process/Organization Matrix, Process/System Matrix, System/Entity Matrix, System/Organization Matrix	[IBM#84]
IDEF, Integration Definition	strukturiert	IDEF0, IDEF1, IDEF3	[RoSc77; FIPS93a; FIPS93b]
SA/SD, Structured Analysis and Design	strukturiert	Data Flow Diagram, RT Data Flow Diagram, Entity Relationship Diagram, Structure Chart, State Transition Diagram	[YoCo79; GaSa79; Your89; McPa84]
OOA/OOD, Object-Oriented Analysis and Design	objektorientiert	Object Diagram, State Transition Diagram, Service Chart	[CoYo91a; CoYo91b]
OMT, Object Modeling Technique	objektorientiert	Class Diagram, Data Flow Diagram, State Transition Diagram	[Rum*91]
OODA, Object Oriented Design with Applications	objektorientiert	Class Diagram, State Transition Diagram, Object Diagram, Module Diagram, Process Diagram	[Booc94]
UML, Unified Modeling Language	objektorientiert	Class Diagram, Use Case Diagram, Collaboration Diagram, Sequence Diagram, State Diagram, Activity Diagram, Component Diagram, Deployment Diagram, Package Diagram	[RuJB99; BoJR99]

Für das systematische Vorgehen des Entwicklers bei der Erstellung einer Anforderungs- bzw. Architekturmodellen für ein Softwaresystem ist in den letzten 20 Jahren eine schier unüberschaubare Fülle von *Methoden* entwickelt worden, in denen in der Regel mehrere *Modellierungstechniken* mehr oder weniger kohärent aufeinander abgestimmt sind. Je nach zugrunde liegendem Modellierungsparadigma und -ziel wird zwischen unterschiedlichen Methodentypen differenziert, z.B. strukturierten und objektorientierten Methoden, datenorientierten Methoden oder Methoden für die Geschäftsprozessanalyse. Tab. 1 liefert ohne Anspruch auf Vollständigkeit einen Überblick über die prominentesten Vertreter der unterschiedlichen Methodengattungen und ihre jeweiligen Modellierungstechniken.

Elemente von Methoden

Obwohl der Begriff der Methode in der Softwaretechnik-Literatur nicht klar umrissen ist und häufig in unterschiedlichen Bedeutungen verwendet wird, werden in den meisten Publikationen zu diesem Thema drei konstituierende Elemente einer Methode genannt (vgl. z.B. [HoVe97; Oll*91; Tolv98; Jar*98; Lyy*98; HaSa96]):

- **Ontologie**: Darunter versteht man die einer Methode zugrunde liegende konzeptuelle Struktur, d.h. den Vorrat an Konzepten und Strukturierungskonstrukten, die zur Beschreibung des Systems verwendet werden. Kernkonzepte im Bereich der Strukturierten Analyse-Methoden [YoCo79; Your89; McPa84; DeMa79] sind z.B. *Prozess*, *Datenfluss* und *Datenspeicher* sowie die hierarchische *Dekomposition* von Prozessen, während bei den objektorientierten Methoden (z.B. [Rum*91; JCJÖ92; Booc94; BoJR99]) die Begriffe *Klasse*, *Objekt* und *Methode* sowie die Strukturierungsprinzipien *Vererbung/Spezialisierung* und *Kapselung* eine beherrschende Rolle spielen.
- **Notation**: Die in einer Ontologie definierte Konzeptwelt kann nur dann verwendet werden, wenn für die einzelnen Konzepte und Strukturierungskonstrukte geeignete Repräsentationen definiert werden. Gerade in Methoden für die frühen Entwicklungsphasen setzen sich grafische Notationen wegen ihrer größeren Verständlichkeit und Übersichtlichkeit gegenüber textuellen durch. Beispiele sind Datenflussdiagramme im Bereich der Strukturierten Analyse oder Klassendiagramme in objektorientierten Methoden. Die Beziehung zwischen den Konzepten und ihrer notationellen Darstellung definiert die Semantik der Notation. Je nach Anwendungszweck ist eine Notation und ihre Beziehung zur zugrunde liegenden konzeptuellen Struktur mehr oder weniger formal definiert. Während Notationen, die mehr auf das menschliche Verständnis und die Kommunizierbarkeit von Systemaspekten ausgelegt sind, eher auf einen rigorosen formalen Unterbau verzichten können, ist eine formale Semantik unerlässlich, um bestimmte Eigenschaften eines mit einer Methode erstellten Systemmodells (Vollständigkeit, Korrektheit, Erreichbarkeit etc.) überprüfen und deduzieren zu können.
- **Prozess**: Neben der Festlegung einer Ontologie und einer Notation ist methodisches Vorgehen am Arbeitsplatz insbesondere an das Befolgen prozeduraler Handlungsrichtlinien und -heuristiken, die mit einer Methode assoziiert sind, geknüpft. Der Prozessaspekt einer Methode gibt vor, wie und in welcher Reihenfolge die gegebenen Konzepte und Notationselemente zur Erstellung eines Systemmodells angewendet werden sollten. Um sinnvolle Resultate zu liefern, müssen sich die Prozessrichtlinien an der konzeptuellen Struktur der Methode orientieren. Zum Beispiel spiegelt sich das Konzept der Dekomposition innerhalb der Strukturierten Analyse in einem Modellierungsprozess wider, der die Top-Down-Verfeinerung des Systemmodells beginnend bei dem Kontextdiagramm vorsieht. Allerdings gibt die konzeptuelle Struktur nur einen groben Rahmen vor, innerhalb dessen eine Reihe unterschiedlicher Vorgehensweisen denkbar sind. Im Kontext der Strukturierten Analyse schlagen beispielsweise McMenamin und Palmer im Gegensatz zu einem strikten Top-Down-Vorgehen die so genannte Ereignispartitionierung vor, bei der zunächst alle Ereignisse, die auf das System einwirken, zu identifizieren sind und danach durch Gruppierung von Stimuli und Systemreaktionen eine Partitionierung des Gesamtsystems vorzunehmen ist [McPa84].

Die methodische Arbeitsplatzunterstützung muss also sowohl Produktaspekte (Ontologie und Notation) als auch Prozessaspekte berücksichtigen. Es fällt auf, dass sich sowohl bei kommerziellen Methoden (z.B. SA, SADT, UML etc.) als auch in der akademischen Forschung über Methodengestaltung (vgl. z.B.

[BrLW96; Brin96; Roll97; Odel96; BrLW96; LyWe99]) das Interesse primär auf den Produktaspekt beschränkt [Wije91; Lyy*98]. In dieser Arbeit werden wir uns mit den Aspekten Ontologie und Notation nur soweit befassen, wie es die Abhängigkeit der Prozesse von den Produkten prinzipiell bedingt, und konzentrieren uns stattdessen auf methodische Unterstützung im Sinne einer Prozessanleitung für die korrekte und effiziente Verwendung von Modellierungskonstrukten einer oder mehrerer Methoden.

Die in der Arbeitsumgebung zur Verfügung stehenden rechnerbasierten Werkzeuge (Editoren, Analysewerkzeuge, Transformatoren etc.), mit deren Hilfe der Entwickler seine vom Projektmanagement zugeteilten Aufgaben unter Verwendung bestimmter Methoden durchführt, spielen eine besondere Rolle für die systematisch-methodische Aufgabendurchführung auf der Arbeitsplatzebene [JaBR99]. Die verfügbare Werkzeugfunktionalität bestimmt im hohen Maße, welche Arbeitsprozesse unterstützt oder überhaupt durchgeführt werden können. So sind der Erstellung und Pflege großer Modelle enge Grenzen gesetzt, wenn dem Entwickler nicht entsprechend leistungsfähige Werkzeuge zur Verfügung stehen. Aus diesem Grund muss die am Arbeitsplatz erteilte Prozessunterstützung für die zu befolgenden Arbeits- und Entscheidungsschritte insbesondere auch auf die zur Durchführung der Aufgaben zur Verfügung stehenden Werkzeuge berücksichtigen.

„Method-Tool-Companionship“ [FoNo92; Tolv98]

Vergleich: Prozessunterstützung auf Projektmanagement- und Arbeitsplatzebene

Im Zentrum dieser Arbeit steht die Unterstützung der *Arbeitsplatzebene*. Um diesen Bereich besser eingrenzen zu können, haben wir in diesem Abschnitt die spezifischen Unterstützungsfunktionen der Arbeitsplatzebene denjenigen der Projektmanagementebene gegenübergestellt. Tab. 2 fasst die wesentlichen Unterschiede der Prozessunterstützung auf Projektmanagement- und Arbeitsplatzebene zusammen.

***Tab. 2**: Prozessunterstützung auf Projektmanagement- und Arbeitsplatzebene*

	Projektmanagementebene	**Arbeitsplatzebene**
Gegenstandsbereich der Prozessunterstützung	Planung, Koordination, Kontrolle des Gesamtprozesses	systematisch-methodische Durchführung einzelner Aktivitäten
Granularität der betrachteten Arbeitseinheiten	grobgranulare Aufgaben, keine Betrachtung der Internstruktur	feingranulare Arbeitsabläufe, Arbeitsschritte unterhalb von Aufgaben
Granularität der betrachteten Entwurfsprodukte	grobgranulare Dokumente als Einheiten der Arbeitsteilung	feingranulare Produkte unterhalb der Dokumentgrenze
Sichtweise auf Entwurfsprodukte	administrative Sichtweise auf Produkte	inhaltsorientierte Sichtweise auf Produkte
Rolle von Werkzeugen	keine oder nur rudimentäre Berücksichtigung von Werkzeugen als Mittel zur Aufgabenerledigung	detaillierte Berücksichtigung von Werkzeugen bei der Durchführung einzelner Arbeitsschritte
Planbarkeit der zu unterstützenden Prozesse	globale und durchgängige Präskription entlang eines Projektplans, jedoch keine Aussagen über detaillierte Ausgestaltung der einzelnen Aufgaben	innerhalb von Aufgaben i.a. kreative Arbeitsprozesse, daher nur fragmentweise, nicht notwendigerweise zusammenhängende Prozessunterstützung für wohlverstandene Teilabläufe

Gegenstandsbereich

Ein primäres Unterscheidungsmerkmal ist der Gegenstandsbereich der Prozessunterstützung auf den beiden Projektebenen. Während auf Projektmanagementebene Unterstützung für die Planung, Koordination und Kontrolle des Gesamt-

prozesses benötigt wird, tritt auf der Arbeitsplatzebene die Anleitung des Entwicklers bei der Durchführung einzelner Aufgaben in den Vordergrund.

Granularität der Arbeitseinheiten

Folglich unterscheidet sich die *Granularität der betrachteten Arbeitseinheiten* auf den beiden Projektebenen. Während auf der Projektmanagementebene die im Entwicklungsprozess auftretenden Aufgaben nur soweit dekomponiert werden, wie es für die Koordination unterschiedlicher Entwickler und die Definition von Meilensteinen erforderlich ist, liegt auf der Arbeitsplatzebene der Fokus auf der Unterstützung feingranularer Arbeitsabläufe. Beispielsweise könnte eine vom Projektmanagement definierte Aufgabe lauten, ein Klassendiagramm für ein bestimmtes (Teil-)System zu spezifizieren, während sich der Entwickler auf der Arbeitsplatzebene dafür interessiert, welche Optionen ihm bei der Verfeinerung einer Klasse zur Verfügung stehen (z.B. Hinzufügen eines diskriminierenden Attributs oder Bildung einer Subklasse).

Granularität der Entwurfsprodukte

Als Folge der unterschiedlichen Granularität der betrachteten Arbeitsschritte differiert auch die *Granularität der Entwurfsprodukte*, die durch die Prozessunterstützung berücksichtigt werden. Während auf der Projektmanagementebene Produkte typischerweise nur auf Dokumentebene als Einheiten der Arbeitsteilung referenziert werden (z.B. ein Klassendiagramm), muss Prozessunterstützung auf der Arbeitsplatzebene auch auf einzelne Produktteile innerhalb der Internstruktur von Dokumenten (z.B. einzelne Klassen oder Attribute innerhalb eines Klassendiagramms) Bezug nehmen.

Sichtweise auf Entwurfsprodukte

Damit einher geht eine unterschiedliche *Sichtweise auf die Entwurfsprodukte*. Für den Projektmanager sind in erster Linie *administrative* Eigenschaften eines Produkts von Belang, z.B. der aktuelle Status eines Software-Dokuments (in Bearbeitung, fertiggestellt, geprüft etc.). Während auf der Ebene der Projektadministration von den konkreten Inhalten eines Dokuments im Allgemeinen abstrahiert wird, ist für die Arbeitsplatzebene ein *inhaltsorientierter* Umgang mit den in Bearbeitung befindlichen Produkten kennzeichnend.

Rolle der Entwicklungswerkzeuge

Die für die Prozessdurchführung verwendeten *Entwicklungswerkzeuge* spielen auf den beiden Ebenen eine unterschiedliche Rolle. Während auf Projektmanagementebene Werkzeuge nicht berücksichtigt oder lediglich als eine Ressource zur Aufgabenbearbeitung (ähnlich wie Dokumente) angegeben werden, muss auf Arbeitsplatzebene der Umgang mit einzelnen Werkzeugen und die Nutzung spezifischer Werkzeugfunktionen zur Durchführung von Arbeitsschritten detailliert in die Prozessunterstützung einbezogen werden.

Planbarkeit der Prozesse

Erhebliche Unterschiede weisen Projektmanagement- und Arbeitsplatzebene hinsichtlich der *Planbarkeit* der zu unterstützenden Prozesse auf. Auf der Projektmanagementebene wird die durchgängige Unterstützung des gesamten Entwicklungsprozesses angestrebt, um ein Instrument für Arbeitsplanung, Ressourcenzuteilung und Fortschrittskontrolle an der Hand zu haben. Unterstützungsansätze für das Projektmanagement sehen sich hier u.a. mit dem Problem konfrontiert, dass auf Änderungen der Rahmenbedingungen (z.B. Kürzung der Projektlaufzeiten/des Budgets, Ausscheiden von Mitarbeitern) und unerwartete Probleme bei der Pro-

Durchgängigkeit, Ausnahmebehandlung

jektdurchführung entsprechend flexibel reagiert werden muss. Die Behandlung von Ausnahmesituation in Prozessunterstützungsansätzen für die Projektmanagementebene ist zur Zeit Gegenstand regen Forschungsinteresses (siehe z.B. [CDFG96; CDGM95; Ober96; CaFM99]). Gemeinsam ist allen Ansätzen jedoch, dass eine gewisse Granularität der betrachteten Aufgaben nicht unterschritten

wird, um das Prinzip der durchgängigen Prozessunterstützung nicht aufgeben zu müssen. Die auf der Arbeitsplatzebene betrachteten Prozesse beziehen sich hingegen auf feingranulare, eng umgrenzte lokale (Teil-)Abläufe. Eine durchgängige Präskription aller Arbeitsabläufe ist hier nicht möglich und wird auch gar nicht angestrebt, da ein großer Teil der in der Softwareentwicklung und insbesondere in den frühen Phasen (Anforderungsanalyse, Entwurf) auftretenden Prozesse nicht ausreichend verstanden ist und – mehr noch als auf der Projektmanagementebene – von der Kreativität der Entwickler geprägt ist und von seinen situativen Entscheidungen und seiner Erfahrungen abhängen [Such87]. Das bedeutet aber nicht, dass auf der Arbeitsplatzebene gar keine über die reine Aufgabenbeschreibung hinausgehende Prozessunterstützung möglich wäre. Bestimmte Teilabschnitte der arbeitsplatzlokalen Arbeitsprozesse lassen sich durchaus als wohldefinierte Abfolgen von Arbeits- und Entscheidungsschritten in den unterschiedlichen Entwicklungswerkzeugen beschreiben. Prozessunterstützung auf der Arbeitsplatzebene ist daher als eine Sammlung feingranularer, nicht notwendigerweise zusammenhängender *Prozessfragmente* aufzufassen [RoPB99; Poh*99].

Fragmentarische Prozessunterstützung

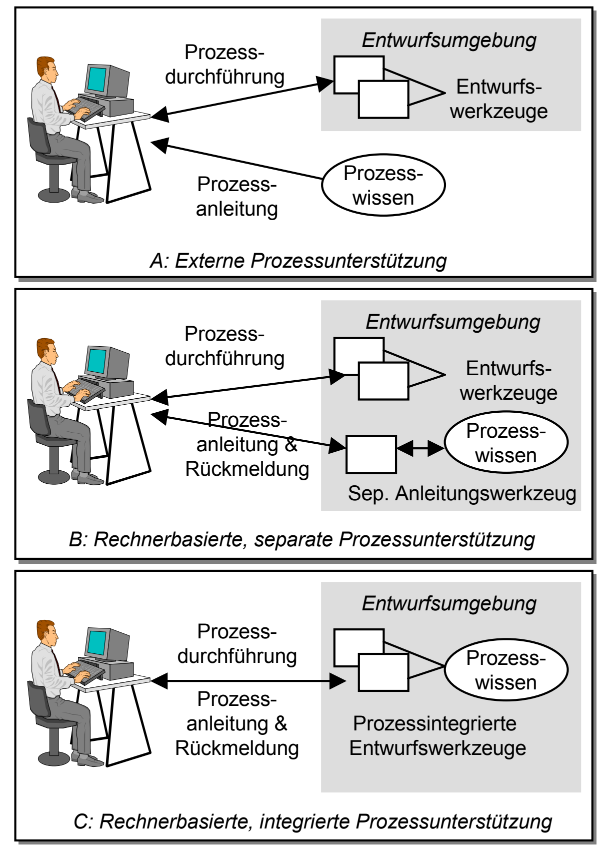
2.1.2 Integrationstiefe

Prozesswissen muss wirkungsvoll kommuniziert werden

Elementare Grundvoraussetzung für die Unterstützung von Entwicklungsaktivitäten ist die wirkungsvolle Kommunikation organisations- und projektspezifischer Handlungsvorgaben und -hilfestellungen an die betroffenen Entwickler am technischen Arbeitsplatz. Um eine ihm zugewiesene Aufgabe korrekt und effizient ausführen zu können, muss der Entwickler mit Wissen über methodische Vorgehensweisen versorgt werden, z.B. worin das Ziel der Aufgabe besteht, welche Ergebnisse zu erbringen sind, aus welchen Teilschritten sich die Aufgabe zusammen setzt, in welcher Reihenfolge die Teilschritte abgearbeitet werden sollen, welche Handlungsalternativen in bestimmten Problemsituationen in Frage kommen bzw. zu favorisieren sind, welche unterstützenden Softwarewerkzeuge zur Verfügung stehen und wie diese zur Durchführung von Teilaufgaben innerhalb des Entwicklungsprozesses sinnvoll eingesetzt werden können.

Die Wirksamkeit der Kommunikation von Prozesswissen hängt wesentlich von der Integrationstiefe der Prozessunterstützung ab. Darunter verstehen wir den Grad der Interaktion und Beeinflussung der übrigen Komponenten der rechnerbasierten Arbeitsumgebung, insbesondere der Entwicklungswerkzeuge, durch die Prozessunterstützung. Generell gilt, dass die Chance, dass die angebotene Unterstützung wahrgenommen wird bzw. dass verbindliche Prozessvorgaben tatsächlich befolgt werden, umso größer ist, je enger die Prozessunterstützung mit den Komponenten der rechnerbasierten Arbeitsumgebung integriert ist. Abb. 2 illustriert den Zusammenhang zwischen dem durch die Prozessunterstützung bereitgestellten Prozesswissen, dem Prozessausführenden (d.h. dem Entwickler) und seiner aus den Entwicklungswerkzeugen bestehenden Arbeitsumgebung. Bezüglich der Integrationstiefe unterscheiden wir drei Ausprägungen: *extern*, *rechnerbasiert-separat* und *rechnerbasiert-prozessintegriert*:

Abb. 2: *Integrationstiefe der Prozessunterstützung mit der rechnerbasierten Arbeitsumgebung*



Externe Prozessunterstützung

Bei der *externen Prozessunterstützung* wird Wissen über die intendierte Prozessausführung unabhängig von der rechnerbasierten Arbeitsumgebung des Entwicklers kommuniziert und tritt dort nicht direkt in Erscheinung. Typische Hilfsmittel, Prozesswissen außerhalb der eigentlichen Arbeitsumgebung zu vermitteln, sind Methoden- und Projekthandbücher (siehe Abschnitt 2.2.1), in denen Arbeitsplatzanweisungen und Projektrichtlinien aufbereitet sind, sowie Trainings- und Schulungsmaßnahmen. Bei der Bearbeitung einer Aufgabe kann daher extern definierte Prozessanleitung nur dann wirksam werden, wenn sich der Entwickler an die Empfehlungen und Anweisungen *erinnert* und entsprechend handelt (siehe Abb. 2a). Andernfalls wird die extern definierte Prozessanleitung nicht genutzt. Zwar können extern vorliegende Prozessanweisungen und -richtlinien durchaus kontextbezogen formuliert sein (vgl. Abschnitt 2.1.3), aber es liegt in der Verantwortung des Entwicklers zu erkennen, welche Arbeitsschritte gemäß den Vorgaben der externen Prozessunterstützung „legal“ und sinnvoll sind und in welchen Situationen sie angewandt werden sollen oder müssen. Zwar können die Prozessausführenden selbstverständlich jederzeit ihr Wissen über Details der intendierten Prozessausführung auffrischen, aber es unterliegt ihrer Initiative, kritische Prozesssituationen als solche zu erkennen und entsprechende Vorgehensempfehlungen und Richtlinien nachzuschlagen.

Rechnerbasierte Anleitung

Der Hauptvorteil rechnerbasierter Ansätze, d.h. *separater* Assistenzwerkzeuge oder *prozessintegrierter* Entwurfswerkzeuge, besteht darin, dass der Entwickler einen im Vergleich zu externer Prozessunterstützung wesentlich direkteren und komfortableren Zugang zu den verfügbaren Prozessdefinitionen erhält. Weiterhin liefern rechnerbasierte Ansätze überhaupt erst die Grundvoraussetzung dafür, dass Änderungen an Prozessdefinitionen ohne großen Aufwand in die Arbeitsumge-

bungen aller betroffenen Prozessausführenden propagiert werden können und dort sofort sichtbar sind (siehe auch Abschnitt 2.1.4).

Separate Anleitungswerkzeuge

Bei *separaten* Assistenzwerkzeugen können Prozessdokumentationen beispielsweise als Informationsnetze aufbereitet sein und durch Hilfesysteme (siehe Abschnitt 2.2.2) oder Webbrowser innerhalb eines organisations- oder projektweiten Intranets für die Entwickler zugänglich gemacht werden. Eine andere Klasse separater Assistenzwerkzeuge repräsentieren so genannte Agenda- oder Taskmanager, die typischerweise das Entwickler-Frontend in prozesszentrierten Umgebungen und Workflow-Managementsystemen darstellen (siehe Abschnitt 2.2.4). Separate Assistenzwerkzeuge sind somit zwar in der Arbeitsumgebung des Entwicklers unmittelbar verfügbar, jedoch mit der eigentlichen technischen Umgebung des Entwicklers, d.h. den für die Aufgabenbearbeitung benötigten, im allgemeinen interaktiven Werkzeuge, nicht oder nur lose gekoppelt (s. Abb. 2b). Der Entwickler muss daher mit einer zusätzlichen Benutzerschnittstelle umgehen, d.h. er nimmt einerseits Empfehlungen, Hinweise und Vorschriften zur Bearbeitung der Aufgabe vom Assistenzwerkzeug entgegen, während er die eigentliche Aufgabe in den interaktiven Entwicklungswerkzeugen erledigt. Als Konsequenz ergibt sich, dass eine mit den Prozessvorgaben konforme Durchführung von Aufgaben auf der technischen Ebene nicht mehr sichergestellt werden kann. Dies ist insbesondere dann kritisch, wenn gemäß der Prozessdefinition in einer bestimmten Situation gewisse Prozessschritte zwingend vorgeschrieben (z.B. die Dokumentation eines Entwurfsobjekts) bzw. verhindert werden sollen (z.B. die Modifikation eines Entwurfsobjekts). Zur Durchsetzung der Prozessvorgaben sind daher zusätzliche organisatorische Maßnahmen der Qualitätskontrolle erforderlich, z.B. die Durchführung von Inspektionen, Reviews und Walkthroughs [Faga76; Faga86; EbSt93; FrWe82; Fowl86; Bias91]

Prozessintegrierte Werkzeuge

Prozessintegrierte Werkzeuge unterstützen den Benutzer, indem sie ihre Arbeitsweise direkt den (projekt- und organisationsspezifischen) Prozessvorgaben unterordnen. Prozessdurchführung und -anleitung werden über die gleiche Benutzerschnittstelle, nämlich die der prozessintegrierten Werkzeuge, abgewickelt. Da dem Entwickler in prozessintegrierten Werkzeugen immer nur die Entwurfsprodukte mit entsprechenden Aktionen zur Bearbeitung angeboten werden, die gemäß der Prozessvorgabe zur Erreichung des aktuellen Prozessziels zweckdienlich bzw. zulässig sind, wird der intendierte Arbeitsprozess direkt in den eigentlichen Entwurfswerkzeugen sichtbar und wirksam (siehe Abb. 2c). Der Entwickler wird somit von der Aufgabe entlastet, die gemäß den Prozessvorgaben in einer Prozesssituation zulässigen Aktionen selbst zu bestimmen. Bestimmte Routineaufgaben lassen sich als komplexe Sequenzen von Werkzeugaktionen sogar automatisieren (siehe auch Abschnitt 2.1.5).

Wir wollen an dieser Stelle nur die generelle Unterscheidung zwischen externer, rechnerbasiert-separater und rechnerbasiert-prozessintegrierter Prozessunterstützung vornehmen. Konkrete Ausprägungen von Prozessunterstützungssystemen unterschiedlicher Integrationstiefe werden wir bei der Untersuchung existierender Ansätze in Abschnitt 2.2 noch detaillierter betrachten.

2.1.3 Kontextbezogenheit

Ein wichtiges Kriterium für die Güte eines Prozessunterstützungssystems ist seine *Kontextbezogenheit*. Kontextbezogenheit ist ein Maß dafür, wie stark eine Unterstüt-

zungsleistung zum Zeitpunkt der Anforderung durch den Entwickler bzw. des Eingreifens in den Arbeitsprozess auf die aktuelle Prozesssituation zugeschnitten ist. Wir unterscheiden hier *statische* Prozessunterstützungsansätze, die unabhängig von der aktuellen Prozesssituation stets die gleiche Unterstützung anbieten, und *dynamische* Prozessunterstützungsansätze, die die tatsächliche Prozesssituation berücksichtigen und die im aktuellen Kontext irrelevanten oder nicht anwendbaren Teile der Prozessunterstützung herausfiltern bzw. eine allgemeine Assistenzfunktion konkretisieren.

Bei der Differenzierung zwischen statischer und dynamischer Prozessunterstützung ist es weniger entscheidend, dass ein Prozessunterstützungsansatz prinzipiell Vorgehenswissen bezüglich unterschiedlicher Prozesssituationen bereithält. Vielmehr interessiert, inwieweit das Unterstützungssystem den Benutzer von der Last befreit, die aktuelle Prozesssituation mit den Randbedingungen der Prozessvorgaben durch die Prozessunterstützung in Beziehung zu setzen und die Anwendbarkeit bestimmter Handlungsanweisungen zu erkennen. Beispielsweise können in einem Projekthandbuch sehr detaillierte Handlungsanweisungen für die unterschiedlichsten Problemsituationen aufgelistet sein (siehe Abschnitt 2.2.1). Dennoch betrachten wir Handbücher als statische Prozessunterstützungsansätze, da es ausschließlich in der Verantwortung des Entwicklers liegt, die aktuell anwendbaren Prozessdefinitionen bestimmen. Im Unterschied dazu sprechen wir bei einem kontextsensitiven Hilfesystem von einem dynamischen Unterstützungssystem (siehe Abschnitt 2.2.2), da dieses bei seinen Erklärungen den aktuellen Prozesskontext, z.B. die Dialogsituation, in der sich der Benutzer in der Interaktion mit der Entwicklungsumgebung befindet, berücksichtigt.

Charakterisierung des Prozesskontexts

Zur Charakterisierung der aktuellen Prozesskontexts können eine ganze Reihe von Merkmalen herangezogen werden. Das primäre Merkmal zur Bestimmung des Prozesskontexts ist der aktuelle Zustand des in Bearbeitung befindlichen Entwicklungsprodukts. Hier kann sich der Zustand auf inhaltlich-strukturelle Eigenschaften beziehen, z.B. ob ein SA-Modell syntaktischen Bedingungen (keine unverbundenen Datenflüsse) und heuristisch begründeten Richtlinien (nicht mehr als sieben Prozesse pro Datenflussdiagramm) genügt, auf inhaltlich-semantische Eigenschaften, z.B. ob ein ER-Modell die in für eine Anwendung relevanten Daten adäquat repräsentiert, oder auf administrative Attribute, z.B. ob ein Dokument einen Review erfolgreich passiert hat oder ob es sich in Überarbeitung befindet. Neben rein produktbezogenen Zustandsattributen spielen zur Bestimmung des aktuellen Prozesskontexts auch die bisherige Prozesshistorie, die sich zum Teil natürlich im Produktzustand widerspiegelt, und die aktuell verfolgten Prozessziele eine entscheidende Rolle. Außerdem rechnen manche Ansätze auch den Erfahrungsstand des Entwicklers, der die Prozessunterstützung anfordert, zum Prozesskontext. Hilfesysteme werden beispielsweise danach unterschieden, ob sie uniforme, d.h. für jeden Entwickler einheitliche, oder individuelle, d.h. auf die spezifischen Fähigkeiten des Entwicklers zugeschnittene Unterstützungsleitungen erbringen [Balz96; BaSc88]. Wir werden in Abschnitt 5.3 noch genauer auf die zur Charakterisierung eines Prozesskontextes relevanten Aspekte eingehen, wenn wir den in dieser Arbeit verfolgten kontextbasierten Prozessmodellierungsansatz vorstellen.

2.1.4 Anpassbarkeit

Prozessverbesserung erfordert Anpassbarkeit

Eine wichtige Voraussetzung für die Akzeptanz eines Prozessunterstützungsansatzes ist seine Anpassbarkeit an organisations- und projektspezifische Gegebenheiten. Das Wissen über Entwicklungsprozesse ist heutzutage selbst in vergleichsweise gut verstandenen Anwendungsdomänen noch lückenhaft und befindet sich im ständigen Fluss, nicht zuletzt wegen der Proliferation der technischen Möglichkeiten. Prozessverbesserungsparadigmen wie Total Quality Management [Demi86], TAME [BaRo88], das Capability Maturity Model des Software Engineering Institutes [Hump89; PCCW93], BOOTSTRAP [Koch93] und SPICE [Dorl93] zielen darauf ab, Ist-Prozesse kontinuierlich auf Schwachstellen hin zu analysieren und zu bewerten, Verbesserungspotenziale zu entdecken und diese in künftige Prozesse einfließen zu lassen (für eine Übersicht über die unterschiedlichen Prozessverbesserungsansätze siehe z.B. [ThMa97]).

Kontingenzfaktoren

Dass es einen allgemeingültigen und durchgängigen Prozess, der den Anforderungen potenziell beliebiger Organisations- und Projektformen genügt, aufgrund der vielfältigen Unwägbarkeiten bei der Systementwicklung nicht geben kann, ist einsichtig und wird auch von vielen Autoren betont (vgl. [CuKO88; Sol#83; KuWe92; AvFi88]). Dies gilt sowohl für projektweite Prozesse, die auf der Projektmanagementebene betrachtet werden, als auch für arbeitsplatzlokale Prozesse auf der technischen Ebene. Zu den so genannten Kontingenzfaktoren [KaRo74; SlBr93], die ganz wesentlich die Auswahl und Ausgestaltung von Methoden und damit auch der Prozessunterstützung beeinflussen, gehören u.a. [SlBr93; Harm97; Oll*91; SlHo96]: Projekttyp, Projektgröße, Wissen und Erfahrung der Mitarbeiter, Innovativität und Komplexität des zu entwickelnden Systems, Klarheit und Stabilität von Anforderungen, Bedeutung des Projekts für die Organisation, Auswirkung auf Geschäftsprozesse etc. Darüber hinaus identifiziert Madhavji weitere Gründe, die zu einer Anpassung existierender Prozesse führen können [Madh91]:

- Der Prozess ist fehlerhaft;
- Der Prozess ist lückenhaft, d.h. es fehlen wichtige Schritte;
- Der Prozess ist zu generisch und muss konkretisiert werden, um im aktuellen Anwendungskontext spezifische Ergebnisse hervorzubringen;
- Die Annahmen, unter denen ein Prozess entworfen wurde, sind durch eine Änderung der Randbedingungen nicht länger gültig;

Wir wollen an dieser Stelle nur die durch Organisations- und Projektspezifika induzierte Notwendigkeit einer anpassbaren Prozessunterstützung festhalten und nicht detailliert auf die Interdependenzen zwischen Kontingenzfaktoren und Entwicklungsprozessen eingehen. Eingehendere Betrachtungen darüber sind in der Literatur über Softwaretechnik (z.B. [Somm92; Nagl96; Pres97; Scha96]), Software- bzw. Projektmanagement (z.B. [Boeh84; Ande90; Car*93; Sarl92; Reif93]) und Prozess- und Methodenmodellierung (z.B. [Chr*97; Harm97; SlHo96; HoVe97]) zu finden. Für den Spezialfall von Nachvollziehbarkeits- und Dokumentationsprozessen im Requirements Engineering wurde in [Dömg99] ein umfangreiches Referenzmodell entwickelt, das den Zusammenhang zwischen verschiedenen Projekteigenschaften und Prozessfragmenten zur Aufzeichnung von Nachvollziehbarkeitsinformationen klärt.

In dem Maße, in dem die ein Projekt beeinflussenden Kontingenzfaktoren variieren können, muss also auch die von einem Prozessunterstützungsansatz geleistete Prozessanleitung flexibel anpassbar sein, um eine sinnvolle und hinreichend spezifische Assistenzfunktion auch im Kontext eines konkreten Projekts zu erhalten. Wir unterscheiden drei Ausprägungen der Anpassbarkeit eines Prozessunterstützungsansatzes: fix , konfigurierbar und änderbar.

Fixe Prozessvorgaben

Von *fixen* oder nicht anpassbaren Prozessunterstützungsansätzen sprechen wir, wenn keinerlei Mechanismen für die Anpassung an geänderte Prozessvorgaben vorgesehen sind. Wenn Prozessunterstützung beispielsweise durch Methoden- und Projekthandbücher vermittelt wird, sind der Anpassbarkeit aufgrund der statischen Natur des Mediums Buch enge Grenzen gesetzt.

Konfiguration durch Customizing

Konfigurierbare Prozessunterstützungsansätze stellen Konfigurationsmechanismen zur Verfügung, mit deren Hilfe die angebotenen Assistenzfunktionen innerhalb vorgegebener Grenzen auf spezifische Belange zugeschneidert werden können. Dieser Anpassungsvorgang wird häufig als *Customizing* bezeichnet. Charakteristisch für den Konfigurationsansatz ist die Idee eines *Referenzmodells* [BeRS99; BaGl98; Reit98], worunter eine abstrakte Repräsentation von Daten-, Funktions-, Organisations- und Prozessstrukturen für einen bestimmten Gegenstandsbereich (z.B. Informationsmodellierung [Gro*97] oder Anforderungsdokumentation [Dömg99]) verstanden wird. Referenzmodelle dienen zur „effizienten Ableitung von unternehmens- bzw. projektspezifischen Strukturen auf der Grundlage vordefinierter Informationsmodelle“ [Reit98]. Die Anpassung an spezifische Belange erfolgt durch Auswahl und Parametrierung der für den jeweiligen Kontext relevanten Ausschnitte des Referenzmodells. Entscheidend für die Bewertung des Anpassungspotenzials eines Konfigurationsansatzes ist, dass das Referenzmodell den Rahmen für die zur Verfügung stehende Prozessunterstützung absteckt und darüber hinaus gehende Prozessunterstützung zunächst nicht vorgesehen ist.

Änderbarkeit und Erweiterbarkeit

In schwach verstandenen Domänen wie (den frühen Phasen) der Softwareentwicklung, in denen Wissen über „ideale“ Entwicklungsprozesse nur fragmentweise vorliegt und durch kontinuierlichen Erfahrungsgewinn ständigen Verbesserungen unterworfen ist, bieten Konfigurationsansätze immer noch nicht hinreichende Flexibilität. Stattdessen muss ein Prozessunterstützungsansatz *änderbar* sein, d.h. er muss Mechanismen zur Verfügung stellen, mit denen die Unterstützung existierender Prozesse abgeändert werden kann (z.B. die Ergänzung eines Ablaufs zur Informationsmodellerstellung um zusätzliche Dokumentationsschritte (vgl. auch [Dömg99]), aber auch Anleitung für völlig neue Prozesse definiert werden kann. Entscheidend ist also, dass nicht nur eine Auswahl und Konkretisierung innerhalb einer Menge vordefinierter Prozesse möglich ist, sondern dass die Prozessunterstützung insbesondere erweiterbar ist.

Prozessunterstützung in CASE-Umgebungen heute noch zu starr

Zusammenfassend sollte ein Prozessunterstützungsansatz also Mechanismen zur Verfügung stellen, mit denen ohne unvertretbar hohen Aufwand die Prozessunterstützung an organisations- und projektspezifische Besonderheiten und an Prozessverbesserungen, die sich im Laufe der Zeit ergeben, angepasst werden können. Angesichts der immer noch mangelnden Ausreifung von Softwareentwicklungsprozessen laufen inflexible Prozessunterstützungsansätze Gefahr, das Experimentieren mit neuen Prozessen prohibitiv teuer zu machen und so langfristig in Hinblick auf die kontinuierliche Prozessverbesserung kontraproduktiv zu wirken [MiSc92; Dene93]. Allgemein wird jedoch beklagt, dass kommerzielle CASE-Umgebungen zu starre Prozessvorgaben machen und wenig Spielraum für

Anpassungen und Erweiterungen lassen [Dene93; JaHu98; AnGr94; EmFi96]. Die Hersteller von CASE-Umgebungen orientieren sich häufig eng an den Vorgaben populärer Methodenhandbücher (z.B. Rational Rose für UML [JaBR99; BoJR99]) und kodieren diese fest in ihre Werkzeuge. Anpassungen an organisations- und projektspezifische Bedürfnisse sind dann nicht mehr oder nur in sehr eingeschränktem Maße möglich. Diese Inflexibilität ist unter anderem ein Grund dafür, warum CASE-Werkzeuge bis heute von vielen Entwicklern eher ablehnend betrachtet werden und den seit Anfang der 90er Jahre immer wieder prognostizierten Durchbruch immer noch nicht ganz geschafft haben [Roth93; Iiva96; JaHu98].

Referenzmodelle in kreativen Entwurfsdomänen kaum verfügbar

Während Referenzmodelle bei der Konfiguration betrieblicher Standardanwendungen bereits seit längerem mit großem Erfolg angewendet werden [ApRi99; BeMS99], wird dieser Anpassungsansatz für kreative Entwurfsprozesse in technischen Domänen bislang noch kaum genutzt bzw. ist gerade erst im Entstehen begriffen sind (z.B. der Unified Process [JaBR99; Kruc98] für die objektorientierte Softwareentwicklung). Als wesentlichen Grund dafür kann man anführen, dass in einem hochkreativen und spezialisiertem Bereich wie der Softwareentwicklung der Aufbau eines etablierten Fundus gutverstandener und umfassender Prozesse wesentlich schwieriger ist als bei betrieblichen Standardanwendungen.

Die Neudefinition und Erweiterung von Prozessen tritt also bei Softwareentwicklungsprozessen wesentlich stärker in den Vordergrund als die Auswahl aus existierenden Prozessen. Als Grundvoraussetzung dafür müssen Konzepte und Mechanismen für die explizite Modellierung von Prozessen und die Interpretation von Prozessmodellen zur Verfügung stehen.

Abschließend wollen wir noch darauf hinweisen, dass Änderungen an Prozessen, die durch die Adaptionsmechanismen eines Prozessunterstützungsansatzes umgesetzt werden sollen, zum Teil weitreichende Auswirkungen auf die produktorientierten Komponenten einer Entwurfsumgebung haben [Lyy*98]. Wenn beispielsweise der Entwicklungsprozess von strukturierter Analyse auf eine objektorientierte Methodik umgestellt werden soll, resultiert diese Änderung nicht nur in veränderten Arbeitsprozessen, sondern muss natürlich auch in den verwendeten Werkzeugen berücksichtigt werden. Konkret bedeutet dies, dass existierende Werkzeuge für die Erstellung von Datenfluss- und ER-Diagrammen durch objektorientierte Spezifikationswerkzeuge ersetzt werden müssen. Produktorientierte Anpassungsmechanismen sind Gegenstand der Forschung im Bereich MetaCASE-Umgebungen oder CASE-Shells (z.B. MetaEdit+ [KeLR95]; Kogge [EbSU97]; MetaView [SoTM88]). Diese Umgebungen bieten Metamodell-basierte Interpretations- und Generierungsmechanismen, mit deren Hilfe Werkzeugumgebungen an geänderte oder neue Modellierungskonzepte und -notationen angepasst werden können. Innerhalb der vorliegenden Arbeit werden wir diesen Typ von Anpassung einer Umgebung jedoch nicht weiter betrachten.

2.1.5 Unterstützungsmodi und Durchsetzungsgrad

Prozessunterstützungssysteme können in die eigentliche Prozessdurchführung auf unterschiedliche Art und Weise eingreifen. Auf der einen Seite können die Handlungen der Prozessausführenden dergestalt eingeschränkt oder gar automatisiert werden, dass die Konformität der Prozessdurchführung mit dem Prozessvorgaben des Prozessunterstützungssystems garantiert wird. In diesem Fall spricht man von einem sehr strikten Durchsetzungsgrad (*enforcement level*) der Prozessunterstützung.

Auf der anderen Seite kann sich die Prozessunterstützung auf das Erteilen von Ratschlägen beschränken, die aus dem aktuellen Prozesszustand deduziert werden. Innerhalb des Spektrums der möglichen Arten der Prozessunterstützung unterscheidet man im allgemeinen vier verschiedene Unterstützungsmodi [DoFe94; GaJa96a; Fern93; Pohl96]: passive Prozessberatung, aktive Prozessanleitung, Prozesslenkung und Prozessautomation.

Passive Prozessberatung

Passive Prozessberatung liefert dem Prozessausführenden auf Anfrage Informationen darüber, welche Handlungen er ausführen sollte, um den Prozessvorgaben der Prozessunterstützung konform zu bleiben. Eine extrem eingeschränkte Form der passiven Prozessanleitung wäre beispielsweise die Bereitstellung eines Handbuchs der organisationsweiten Entwicklungsrichtlinien und -prozeduren. Unter Zuhilfenahme von Informationen über den aktuellen Prozesszustand kann die passive Prozessberatung kontextsensitive Ratschläge geben, die den aktuellen Prozesszustand berücksichtigen, z.B. eine Liste der als nächstes möglichen Arbeitsschritte (siehe Abschnitt 2.1.3). Andere Formen der passiven Prozessberatung bestehen darin, dem Prozessausführenden einen Überblick über den aktuellen Prozesszustand (z.B. Bearbeitungszustand der einzelnen Aufgaben und Dokumente) zu geben oder die Auswirkung hypothetisch ausgeführter Aktionen auf den Prozesszustand untersuchen zu können (z.B. welche Anpassungsoperationen eine Änderung eines Produkts nach sich zieht). Wesentlich für die passive Prozessberatung ist, dass sie nur auf Initiative des Prozessausführungen in Aktion tritt und dass im weiteren Prozessverlauf die Adäquatheit der Prozessunterstützung nicht unbedingt davon abhängt, dass die zuvor angebotene Unterstützung tatsächlich angenommen und befolgt wird.

Aktive Prozessanleitung

Aktive Prozessanleitung basiert wie die passive Prozessberatung auf Rückmeldungen bzgl. der aktuellen Prozessdurchführung, so dass die Prozessmodellausführung dem Prozessausführenden Ratschläge oder Information geben kann, die im aktuellen Prozesszustand relevant sind. Im Gegensatz zur passiven Beratung tritt die aktive Anleitung nicht erst auf Anfrage des Prozessausführenden in Aktion, sondern greift in bestimmten Situationen aktiv in den Prozess sein. Aktive Anleitung kann darin bestehen, neue Aufgaben in die Agenda eines Prozessausführenden einzufügen, ihn über bestimmte Ereignisse in Kenntnis zu setzen (z.B. dass ein von ihm benötigtes Dokument von einem anderen Teammitglied freigegeben wurde) oder Warnmeldungen auszugeben (z.B. dass die Deadline für die Erfüllung einer Aufgabe näherrückt).

Prozesslenkung

Bei der *Prozesslenkung* wird die eigentliche Prozessausführung so geführt, dass sie mit den Prozessvorgaben der Prozessunterstützung konform bleibt. Effektiv ist dies nur durch strikte Kontrolle des Benutzerzugriffs auf Daten und Werkzeugdienste möglich. Dies kann indirekt geschehen, indem der Zugriff auf eine Teilmenge der in einer Umgebung verfügbaren Daten und Werkzeugdienste eingeschränkt wird, oder direkt, indem die im aktuellen Prozesszustand relevanten und ggf. interaktiven Werkzeugdienste auf geeigneten Datenobjekten von der Prozessmaschine initiiert werden.

Prozessautomation

Prozessautomation bezeichnet die automatische Durchführung eines Teils des Prozesses unter Kontrolle der Prozessunterstützung. Prozessautomation ist immer dann angemessen, wenn komplexe, stereotype Abfolgen von Entwicklungsschritten ohne Benutzerintervention und -entscheidung von nichtmenschlichen Agenten, d.h. Werkzeugen, abgewickelt werden können. Ein guter Kandidat für Prozessautomation ist beispielsweise der Buildprozess (d.h. die Aktualisierung des

Codes eines ausführbaren Programms nach der Änderung eines oder mehrerer Quelldateien), der durch ein Make-Skript spezifiziert und gesteuert wird.

Unterstützungsmodi ergänzen sich

In vielen Publikationen wird betont, dass sich die einzelnen Unterstützungsmodi nicht gegenseitig ausschließen, sondern synergetisch ergänzen [Heim90; BaDF96; Pohl96; DoFe94; WALM99]. Daher sollte dem Entwickler in einer Entwurfsumgebung das gesamte Unterstützungsspektrum von passiver Prozessberatung bis hin zu Prozessautomation zur Verfügung stehen. Auf welche Art die Prozessunterstützung in den Entwicklungsprozess eingreifen sollte, hängt in hohem Maße von der jeweils zu unterstützenden Aktivität ab. Generell gilt die Softwareentwicklung, aber auch Entwurfstätigkeiten in anderen Bereichen (z.B. Verfahrenstechnik), als ein offener, iterativer und inkrementeller Problemlösungsvorgang, der sich einer durchgehenden und umfassenden feingranularen Präskription entzieht. Aktivitäten, die durch einen hohen Anteil kreativer Entscheidungssituationen gekennzeichnet sind oder deren Ergebnisse sich nur schwer vorhersagbar sind, lassen sich somit nur schwer präskriptiv unterstützen [Lehm87; Lehm91]. In diesem Fall stellen passive Prozessberatung und ggf. aktive Prozessanleitung die adäquateren Unterstützungsmodi dar. Eine wichtige Rolle spielt auch der Erfahrungsstand des Prozessausführenden. Während unerfahrene Entwickler eher darauf angewiesen sind, aktiv durch den Prozess geführt zu werden, kann eine zu restriktive Prozesslenkung bei Experten schnell zu Produktivitätsverlusten und damit zu einer ablehnenden Haltung gegenüber der Prozessunterstützung führen.

Dennoch lassen sich sogar in den frühen Phasen der Softwareentwicklung (Anforderungsdefinition, Entwurf) durchaus wohlverstandene Routineaktivitäten mit repetitivem Charakter identifizieren, die von einem lenkendem oder gar automatisierendem Prozessunterstützungsansatz profitieren können [Pohl96]. Eine strikte Einflussnahme der Prozessunterstützung auf die Prozessausführung ist auch für solche Aktivitäten angebracht, die aufgrund vertraglicher Vereinbarungen nach bestimmten Vorgaben oder Standards ausgeführt werden müssen. In vielen Qualitätsstandards wird beispielsweise die Nachvollziehbarkeit der Entwicklung und Wartung systemkritischer Komponenten gefordert (vgl. z.B. ISO 9000-3 [ISO#91], das V-Modell [BrDr95] oder den DoD Standard 2176A [DoD#88]). Um dies zu gewährleisten, könnte die Prozessunterstützung den Prozess beispielsweise dergestalt lenken, dass der Entwickler begleitend zu jeder Änderung des Systemmodells die zugrunde liegenden Entwurfsentscheidungen zu protokollieren hat [Dömg99].

2.2 Bewertung existierender Ansätze

Wir wollen nun existierende Ansätze zur Prozessunterstützung anhand des im vorangegangenen Abschnitt vorgestellten Kriterienkatalogs klassifizieren und einander gegenüberstellen.

Unter Prozessunterstützung kann man zunächst alle organisatorischen und technischen Maßnahmen verstehen, die zu einer effizienteren Koordination, Kontrolle und Ausführung von Entwurfsprozessen führen. Macht man sich diese noch sehr allgemeine Definition zu eigen, kann man freilich auch die „traditionellen", produktorientierten Komponenten einer Entwurfsumgebung, d.h. CASE-Editoren [Fugg93; Nils89; FoNo92], Transformations- und Integrationswerkzeuge [Lefe95; Nagl96], Entwurfsrepositorien [BeDa94; Tann94; Ortn99; HaLe93; IRDS90;

NiJa99], Konfigurations- und Versionsmanagementsysteme [Katz90; CoWe98], Kommunikationsmechanismen [Reis90; OMG#97; Schi93; OrHE96] und andere Infrastrukturkomponenten, zur Prozessunterstützung innerhalb einer Entwurfsumgebung hinzurechnen, da die bloße Verfügbarkeit solcher Dienste bereits einen erheblichen Einfluss auf Softwareentwicklungsprozesse hat [JaBR99]. So wären beispielsweise Anforderungsdefinitionsprozesse, in denen Anforderungsspezifikationen mit nicht selten Hunderten von (grafischen) Modellen entstehen, ohne Zuhilfenahme moderner CASE-Werkzeuge, Entwurfsrepositorien sowie Konfigurations- und Versionsmanagementwerkzeuge zur Verwaltung der entstehenden Entwurfsdaten schlichtweg undenkbar.

Wir wollen uns im Folgenden aber auf solche Unterstützungskonzepte konzentrieren, die über die reine Produkterstellung und -verwaltung, die ja immer Teil der Prozessunterstützung ist, hinausgehen und den Prozessaspekt besonders betonen. Dazu zählen wir insbesondere die effektive Kommunikation von Wissen über erprobte und anzuwendende Vorgehensweisen an die Prozessbeteiligten und die problembezogene und zielgerichtete Anleitung, Lenkung oder Automation von Entwicklungsaktivitäten. Als in heutigen Organisationen mehr oder weniger geläufige Ansätze zur Prozessunterstützung betrachten wir *Methoden- und Projekthandbücher* (Abschnitt 2.2.1), *Hilfesysteme* (Abschnitt 2.2.2), *Assistenten/Interface-Agenten* (Abschnitt 2.2.3) und *prozesszentrierte Entwicklungsumgebungen* (Abschnitt 2.2.4).

2.2.1 Methoden- und Projekthandbücher

Die in der heutigen industriellen Praxis immer noch am weitesten verbreitete Methode, Prozesswissen zu kommunizieren, besteht darin, allgemeine Handlungsrichtlinien und Softwaretechnik-Standards (z.B. ISO 12207 [ISO#95], IEEE 1074 [IEEE95], PSS-05 [Maz*94], V-Modell [BrDr95; DrHM98]), methodenbezogene Vorgehensmodelle (z.B. UML Unified Process [JaBR99; Kruc98], ICONIX Unified Object Modeling Approach [RoSc99]) und organisations- und projektspezifische Arbeitsplatzanweisungen in Form von *Handbüchern* aufzubereiten. Die Verbreitung des in den Handbüchern hinterlegten Prozesswissens wird im Allgemeinen durch Trainingsprogramme und Schulungen begleitet, in denen die Prozessausführenden entsprechend den Vorgaben aus den Handbüchern für die ihnen zugewiesenen Aufgaben qualifiziert werden.

***Tab. 3**: Prozessunterstützung durch Methoden- und Projekthandbücher*

Art der Prozessunterstützung	**Projektebene**		**Integrationstiefe**			**Kontextbezogenheit**		**Anpassbarkeit**			**Unterstützungsmodi**			
	administrativ	technisch	extern	rechnerbasiert separat	rechnerbasiert integriert	statisch	dynamisch	fix	konfigurierbar	änderbar	passive Beratung	aktive Anleitung	erzwingend, lenkend	automatisierend
Methoden- und Projekthandbücher	+	+	+	–	–	+	–	+	–	–	+	–	–	–

Tab. 3 zeigt die Einordnung der Prozessunterstützung durch Projekt- und Methodenhandbücher in den Klassifikationsrahmen:

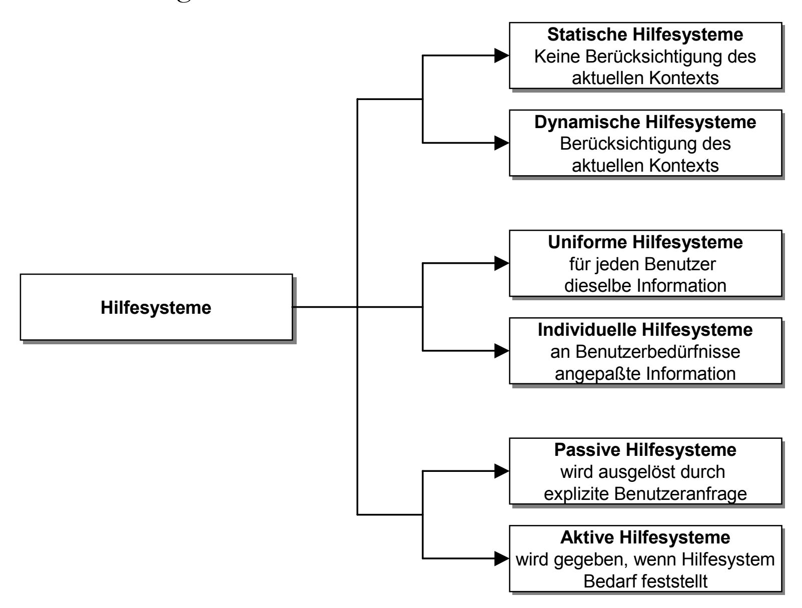
- **Unterstützte Projektebene**: Das in Projekt- und Methodenhandbüchern hinterlegte Prozesswissen kann sowohl die Projektmanagement- als auch die Arbeitsplatzebene betreffen.
- **Integrationstiefe**: Bei der Prozessunterstützung durch Handbücher handelt es sich um eine Form der *externen Prozessunterstützung*, d.h. die in einem Handbuch definierte Prozessanleitung ist nicht direkt in der rechnerbasierten Arbeitsumgebung des Entwicklers zugänglich und sichtbar. Bei der Bearbeitung einer Aufgabe kann die im Handbuch definierte Prozessanleitung nur dann wirksam werden, wenn sich die Prozessausführenden an die Empfehlungen und Anweisungen erinnern und entsprechend handeln. Andernfalls wird die im Handbuch definierte Prozessanleitung nicht genutzt. In der Praxis bedeutet dies, dass Projekt- und Methodenhandbücher häufig ignoriert werden und eher als bürokratischer Ballast, denn als wirkliche Hilfe angesehen werden [Bec*99].
- **Kontextbezogenheit**: Die durch Handbücher geleistete Prozessunterstützung ist *statisch*. Es liegt in der Verantwortung des Entwicklers zu erkennen, welche Arbeitsschritte gemäß den Vorgaben aus dem Handbuch „legal“ und sinnvoll sind und in welchen Situationen sie angewandt werden sollen oder müssen. Zwar können die Prozessausführenden selbstverständlich jederzeit ihr Wissen über Details der intendierten Prozessausführung auffrischen, aber es unterliegt ihrer Initiative, kritische Prozesssituationen als solche zu erkennen und entsprechende Vorgehensempfehlungen und Richtlinien in Handbüchern nachzuschlagen und zu befolgen.
- **Anpassbarkeit**: Das in Handbüchern niedergelegte Prozesswissen ist *fix*. Änderungen in organisations- und projektspezifischen Prozessdefinition rufen offensichtliche Probleme hervor: die Handbücher müssen aktualisiert werden, jeder von der Änderung potenziell betroffene Prozessausführende muss von der Änderung informiert werden. Je nach Art der Änderung sind gegebenenfalls erneute Trainingsmaßnahmen und Schulungen erforderlich.
- **Unterstützungsmodi**: Handbücher können lediglich *passive* Prozessunterstützung anbieten. Ein aktives Eingreifen in den Arbeitsprozess ist wegen der nicht vorhandenen Kopplung mit den Komponenten einer rechnerbasierten Entwurfsumgebung nicht möglich.

2.2.2 Hilfesysteme

Hilfesysteme sind rechnergestützte Systeme, die den Benutzer durch explizite Erklärungen und Auskünfte beim Umgang mit der Mensch-Computer-Schnittstelle von Werkzeugen und Anwendungssoftware unterstützen [Shne98; Balz96; EbOO94; Wand93; Herc94]. Hilfesysteme werden in der Regel als Bestandteil von Anwendungsprogrammen und Werkzeugen ausgeliefert. Folglich ist das von solchen Hilfesysteme vermittelte Wissen in der Regel stark *werkzeugbezogen* (z.B. Erläuterungen zu Werkzeugfunktionen, Bedeutung von Dialogelementen etc.). Es gibt jedoch eine Reihe von prozessbezogenen Ansätzen, die Prozessdokumentation meist in Web-basierter Form aufbereiten und strukturieren (z.B. *SPEARMINT* [Bec*99], *Electronic Process Guides* [Kel*98], *V-Model Browser* [VeMü97]. Dazu zählen auch so genannte *Process Asset Libraries* (PAL), die seit einigen Jahren von vielen Firmen systematisch zusammengetragen und den Mitarbeitern innerhalb des

Firmen-Intranets zugänglich gemacht werden. Häufig basieren diese Systeme auf *Lotus Notes*, wie zum Beispiel das Produkt *aimfirst* der Firma *aimware*, das die Organisation und Verwaltung von Process Asset Libraries unterstützt.

***Abb. 3**: Klassifikationsmodell für Hilfesysteme (vgl. [BaSc88; Balz96])*



Hilfesysteme variieren in ihrer Unterstützungsleistung sehr stark. Im einfachsten Fall wird nach dem Betätigen einer Hilfetaste in einem Werkzeug nur das entsprechende Kapitel aus dem Benutzerhandbuch des Werkzeugs angezeigt. Weitergehende Hilfesysteme berücksichtigen den aktuellen Arbeitskontext, unterstützen jeden Benutzer individuell und initiieren Hilfeleistungen zum Teil selbstständig ohne Zutun des Benutzers. Abb. 3 zeigt eine Klassifikation von Hilfesystemen (vgl. [BaSc88; Balz96]).

Statische vs. dynamische Hilfesysteme

Zum einen wird sich zwischen *statischen* und *dynamischen* Hilfesystemen unterschieden. Ein statisches Hilfesystem zeigt unabhängig vom aktuellen Arbeitssituation und der tatsächlichen Dialogsituation immer die gleiche Hilfeinformation an. Auf dieselbe Frage des Benutzers gibt es stets die gleiche Antwort, unabhängig vom Kontext. Im Gegensatz dazu berücksichtigt ein dynamisches Hilfesystem bei seinen Erklärungen die spezifische Situation dahingehend, dass aktuell irrelevante Teile der Antwort herausgefiltert werden bzw. eine allgemeine Antwort konkretisiert wird.

Uniforme vs. individuelle Hilfesysteme

Als weiteres Unterscheidungsmerkmal von Hilfesystemen wird manchmal ihre Fähigkeit herangezogen, die Hilfeleistung auf den jeweiligen Benutzer abstimmen zu können. Ein *uniformes* Hilfesystem liefert jedem Benutzer dieselbe Hilfeleistung unabhängig von seinem Kenntnis- und Erfahrungsstand. Eine *individuelles* Hilfesystem unterscheidet dagegen verschiedene Benutzergruppen oder modelliert die Eigenschaften eines jeden Benutzers individuell, um so gezieltere Hilfestellung geben zu können.

Passive vs. aktive Hilfesysteme

Ein *passives* Hilfesystem geht davon aus, dass der Benutzer von sich aus aktiv wird und eine Hilfeleistung, z.B. durch Drücken der Hilfetaste, anfordert und eine Anfrage an das Hilfesystem stellt (z.B. durch Eingabe von Schlüsselwörtern, Navigation durch Informationsnetze, Anfragen in natürlicher Sprache). Ein Benutzer ist immer dann in der Lage, passive Hilfe anzufordern, wenn er selbst Probleme erkennt, bei denen er Unterstützung benötigt oder nach bestimmten Informationen, Funktionen oder Kommandos sucht.

Ein *aktives* Hilfesystem beobachtet das Benutzerverhalten und wird von sich aus aktiv, um dem Benutzer eine Hilfe zu geben. Dies ist besonders dann hilfreich, wenn sich der Benutzer selbst nicht darüber im Klaren ist, dass er in eine Problemsituation geraten ist oder dass es effizientere Wege zur Erledigung einer Aufgabe gibt.

Von einem idealen Hilfesystem wird erwartet, dass es das gesamte Spektrum der skizzierten Hilfearten abdeckt. Tatsächlich handelt es sich jedoch bei den heute in der Praxis eingesetzten Hilfesystemen um passive und uniforme Systeme mit einer Mischung aus statischen und dynamischen Hilfeleistungen [Balz96; Wand93; Shne98].

Tab. 4 ordnet die unterschiedlichen Arten von Hilfesystemen in den Klassifikationsrahmen für Prozessunterstützungssysteme aus Abschnitt 2.1 ein. Zur Vereinfachung der Klassifikation von Hilfesystemen haben wir die Unterscheidung zwischen statischen und dynamischen Systemen einerseits sowie uniformen und individuellen Systemen andererseits aufgehoben. Diese Sichtweise lässt sich dadurch rechtfertigen, dass man die Berücksichtigung von Benutzerprofilen auch als Teil des aktuellen Benutzungskontexts des Hilfesystems betrachten kann. In diesem Sinne ist uniforme Hilfe ein Spezialfall statischer Hilfe, während individuelle Hilfesysteme immer auch dynamische Hilfesysteme darstellen.

- **Unterstützte Projektebene**: Die von einem Hilfesystem bereitgestellte Prozessunterstützung bezieht sich in erster Linie auf den Umgang mit Softwarewerkzeugen und Anwendungsprogrammen, indem z.B. Informationen über die im aktuellen Kontext wählbaren Objekte und Funktionen, Erklärungen und Hinweise zu Eingabeaufforderungen, Erläuterungen zu Ergebnissen von Funktionsausführungen und weiterführende Erklärungen zu Fehlermeldungen geliefert werden. Diese Art werkzeugbezogenen Prozesswissens adressiert in der Regel feingranulare, eng umrissene Prozessschritte aus Sicht des technischen Entwicklers. Daher kommt die durch Hilfesysteme geleistete Prozessunterstützung in erster Linie der technischen Arbeitsplatzebene zugute, obwohl Hilfesysteme prinzipiell auch die Benutzung von Werkzeugen des Projektmanagements unterstützen können.

- **Integrationstiefe**: Hilfesysteme sind ihrer Definition nach rechnerbasierte Systeme. Sie sind im Allgemeinen von den eigentlichen Werkzeugen der Entwicklungsumgebung entkoppelt in dem Sinne, dass ein Hilfesystem nicht direkt auf das Verhalten von Entwicklungswerkzeugen Einfluss nehmen kann, sondern dem Benutzer lediglich Empfehlungen und Benutzungshinweise vermitteln kann, die der Benutzer dann selbst in den jeweiligen Werkzeugen umsetzen muss. Im Fall dynamischer bzw. aktiver Hilfesysteme liegt allerdings eine stärkere Kopplung zwischen Hilfesystemen und Entwicklungswerkzeugen vor, da das Hilfesystem mit Informationen über den aktuellen Bearbeitungskontext, z.B. die momentane Dialogsituation in einem Werkzeug, versorgt werden muss. Aber auch bei diesen Typen von Hilfssystemen obliegt die Umsetzung der durch das Hilfesystem geleisteten Prozessanleitung im jeweiligen Werkzeugen dem Benutzer. Werkzeugbezogene Hilfesysteme sind inhaltlich und systemtechnisch eng mit „ihren“ jeweiligen Werkzeugen verknüpft, so dass der Benutzer in einer heterogenen Entwicklungsgebung, die aus einer größeren Zahl von Werkzeugen unterschiedlicher Hersteller bestehen, mit einer ebenso gro-

ßen Zahl untereinander im Allgemeinen nicht integrierter Hilfesysteme umgehen muss. Als Konsequenz bieten Hilfesysteme bei werkzeugübergreifenden Abläufen kaum oder gar keine Prozessunterstützung.

- **Kontextbezogenheit**: Gemäß der oben vorgenommen Klassifizierung von Hilfesystemen stellen dynamische und/oder individuelle Hilfesysteme kontextbezogene Prozessunterstützungsmechanismen dar, während statische und/oder uniforme Hilfesysteme nicht kontextbezogen unterstützen. Passive Hilfesysteme können sowohl statisch als auch kontextbezogen sein. Aktive Hilfesysteme stellen von sich aus fest, wann Hilfe benötigt wird und berücksichtigen somit automatisch den Kontext zum Zeitpunkt der Hilfeleistung.
- **Anpassbarkeit**: Hilfesysteme sind in der Regel nicht anpassbar oder nur grob konfigurierbar, etwa durch Voreinstellung eines Benutzerprofils (z.B. Anfänger, fortgeschrittener Benutzer, Experte). Die Aktualisierung des vom einem Hilfesystem bereitgestellten Prozesswissens geht meistens einher mit der Installation einer neuen Version des zugehörigen Werkzeugs/Anwendungsprogramms. Manche Hilfesysteme erlauben die Annotation von Hilfeseiten mit eigenen Notizen oder die Definition von benutzerspezifischen Lesezeichen innerhalb des Hilfesystems.
- **Unterstützungsmodi**: Gemäß ihrer Definition bieten passive Hilfesysteme nur Prozessberatung an, während aktive Hilfesysteme ohne explizite Benutzeraufforderung anleitend in den Prozess eingreifen. Dies ist unabhängig davon, ob die Hilfeleistung statisch ist oder den aktuellen Kontext und /oder den Erfahrungsstand des Benutzers berücksichtigt. Da Hilfesysteme außer dem Auslesen von Statusinformationen nicht direkt mit dem zugrunde liegenden Werkzeug interagieren, können Hilfesysteme nicht lenkend (in dem Sinne, dass sie den Zugriff auf bestimmte Werkzeugaktionen verhindern) oder gar automatisierend in den Prozess eingreifen.

***Tab. 4**: Prozessunterstützung durch Hilfesysteme*

Art der Prozessunterstützung	Projektebene		Integrationstiefe			Kontextbezogenheit		Anpassbarkeit			Unterstützungsmodi			
	administrativ	technisch	extern	rechnerbasiert separat	rechnerbasiert integriert	statisch	dynamisch	fix	konfigurierbar	änderbar	passive Beratung	aktive Anleitung	erzwingend, lenkend	automatisierend
Hilfesysteme														
statisch, uniform	o	+	–	+	–	+	–	+	o	–	/	/	–	–
dynamisch, individuell	o	+	–	+	o	–	+	+	o	–	/	/	–	–
passiv	o	+	–	+	–	/	/	+	o	–	+	–	–	–
aktiv	o	+	–	+	o	–	+	+	o	–	+	o	–	–

2.2.3 Assistenten

In modernen Entwicklungswerkzeugen und -umgebungen, aber auch in Standardapplikationen wie Textverarbeitungs- und Tabellenkalkulationsprogrammen ist eine stetige Zunahme des Funktionsumfangs zu beobachten. Viele Anwender sind

mit der damit einher gehenden, steigenden Benutzungskomplexität überfordert. Auch die Unterstützungsleistung von Hilfesystemen wird hier häufig als unzureichend empfunden. Zum einen ist das Auffinden der relevanten Informationen nicht immer einfach, da der Benutzer häufig gar nicht genau weiß, nach welchem Hilfethema er suchen soll. Zum anderen müssen die durch das Hilfesystem gelieferten Handlungsanweisungen erst noch in eine Abfolge von manuell anzustoßenden Werkzeugaktionen umgesetzt werden.

Interface-Agenten

Aus der Forschung über intelligente Benutzerschnittstellen stammt das Konzept der *Interface-Agenten* [Maes94; Laur90], welche eine Weiterentwicklung aktiver, dynamischer Hilfesysteme darstellen. Interface-Agenten überwachen laufend das Verhalten eines Benutzers beim Umgang mit einer komplexen Applikation, versuchen Probleme selbstständig zu erkennen und geben gegebenenfalls Hilfestellung über eine einfache, aufgabenbezogene Schnittstelle. In der Praxis werden solche Schnittstellen häufig Assistenten oder „Wizards“ genannt; ein bekanntes Beispiel sind die Microsoft Office-Assistenten [Micr97], die aus den Erkenntnissen des Lumière-Projekts [Hor*98] hervorgegangen sind.

Microsofts Office-Assistenten

Aus Benutzersicht sind Assistenten meist als eine Abfolge von Dialogen realisiert, die schrittweise die für einen Vorgang benötigten Informationen vom Benutzer erfragen und im zugrunde liegenden Werkzeug entsprechende Aktionen auslösen. Als Beispiel sei der mit Microsoft Access 2000 ausgelieferte Tabellenentwurfs-Assistent angeführt. Hier wird der Benutzer durch den Prozess der Erstellung eines Datenbankschemas geführt. Im ersten Schritt wird dem Benutzer eine Liste von Beispieltabellen zur Auswahl angeboten, die er als Grundlage für seinen Tabellenentwurf verwenden kann und aus denen er wiederum die für seinen Zweck relevanten Datenfelder selektieren kann. Im nächsten Schritt kann der Benutzer an allen oder einigen der übernommenen Felder auf seinen Anwendungssachverhalt bezogene Anpassungen vornehmen (z.B. Umbenennung von Feldern). Danach weist der Tabellenassistent den Benutzer an, die Tabelle zu benennen und einen Primärschlüssel auszuweisen oder aber die Generierung eines Primärschlüssels dem Assistenten zu überlassen. Im nächsten Schritt wird der Benutzer angeleitet, Beziehungen zu bereits existierenden Tabellen der Datenbank zu spezifizieren. Schließlich erzeugt der Assistent aus den vom Benutzer erfragten Information die neue Tabelle.

Für jeden der genannten Schritte interagiert der Assistent über jeweils einen hochspezialisierten Dialog (der wiederum gegebenenfalls einen oder mehrere Unterdialoge besitzt) mit dem Benutzer. Das hinter dem angeleiteten Ablauf steckende Prozesswissen manifestiert sich somit in der Dialogabfolge und im Aufbau der einzelnen Dialoge des Assistenten. Um die Benutzungskomplexität gegenüber dem zugrunde liegenden Basiswerkzeug zu reduzieren, bieten Assistenten häufig nicht alle Entwicklungsoptionen, sondern liefern als Entwurfsresultat häufig nur ein Grundgerüst, das vom Benutzer mit den Standardfunktionalitäten des Werkzeugs noch angepasst und erweitert muss. Im obigen Beispiel wird der Benutzer die generierte Tabelle im Allgemeinen um weitere Felder erweitern oder Datentyp- und Formatdefinitionen existierender Datenfelder an seine Bedürfnisse anpassen.

Aufgabenspezifische Dialogführung

Tab. 5 gibt einen Überblick über die Charakteristika der von Assistenten und Interface-Agenten gelieferten Prozessunterstützung.

- **Unterstützte Projektebene**: Die von Assistenten geleistete Unterstützung ist wie bei Hilfesystemen sehr stark auf den Umgang mit bestimmten

Werkzeugen zugeschnitten. Daher profitiert hauptsächlich die technische Arbeitsplatzebene von der Unterstützungsleistung durch Assistenten, da die Arbeitsprozesse auf dieser Ebene wesentlich stärker als auf der Projektmanagementebene durch die Verwendung komplexer Werkzeuge geprägt ist. Prinzipiell können Assistenten allerdings auch Projektmanagementprozesse unterstützen, wenn sie zum Beispiel den Umgang mit einem Projektmanagementwerkzeug vereinfachen.

- **Integrationstiefe**: Assistenten sind im Allgemeinen keine eigenständigen Komponenten einer (aus heterogenen Entwicklungswerkzeugen bestehenden) Entwicklungsumgebung, sondern werden vom jeweiligen Werkzeughersteller als integraler Bestandteil eines komplexen Werkzeugs oder einer eng integrierten Werkzeugsammlung entwickelt und vertrieben. Von daher ist die von einem Assistenten gelieferte Prozessunterstützung sehr eng auf das jeweilige Werkzeug zugeschnitten. Insbesondere steuern Assistenten in den jeweiligen Werkzeugen automatisch bestimmte Teilfunktionen an bzw. fokussieren als meist modale Dialogfenster die Interaktionsmöglichkeiten des Benutzers auf die für den aktuellen Vorgang relevanten Optionen. Ähnlich wie bei Hilfesystemen führt die enge Kopplung eines Assistenten mit „seinem“ Werkzeug dazu, dass innerhalb eines größeren Werkzeugverbunds, aus dem sich eine Entwicklungsumgebung typischerweise konstituiert, jeweils nur werkzeuglokale Unterstützung stattfindet, während Prozesse, die mehrere Werkzeuge involvieren, normalerweise nicht durch Assistenten unterstützt werden können. Ausnahmen bilden sehr eng integrierte Entwicklungsumgebungen, deren Teilwerkzeuge von einem Hersteller stammen, so dass die Grenzen zwischen den Einzelwerkzeugen und der Gesamtumgebung verschwimmen (z.B. Microsofts Visual Studio oder Oracle Designer 2000).

- **Kontextbezogenheit**: Sobald ein Assistent aktiviert wurde, führt er den Benutzer durch den von ihm unterstützten Vorgang. Die dabei geleistete Prozessunterstützung ist in dem Sinne kontextsensitiv, dass der Assistent in Abhängigkeit von bereits durchgeführten Teilschritten, d.h. gemäß aktuellem Prozessfortschritt, eine Auswahl von möglichen nächsten Schritten vorschlägt, den Benutzer nach bestimmten Informationen fragt oder Werkzeugaktionen automatisch anstößt.

- **Anpassbarkeit**: Assistenten verkörpern mehr oder weniger hochspezialisiertes Prozesswissen, das in ihrer Implementierung fixiert wurde, aber nicht auf einer konzeptuellen Ebene zugänglich und anpassbar ist. Im Allgemeinen besteht keine Möglichkeit, die in einem Assistenten hartkodierten Prozessvorgaben entsprechend organisations- und projektspezifischen Bedürfnissen zu konfigurieren (z.B. bestimmte Auswahlmöglichkeiten auszublenden), zu ändern (z.B. die vom Assistenten vorgesehene Abfolge von zwei Arbeitsschritten zu vertauschen) oder zu erweitern (z.B. zusätzliche Arbeitsschritte einzufügen). Wegen der engen Integration von Assistenten mit „ihren“ jeweiligen Werkzeugen bleibt die Anpassung von Assistenten bzw. die Entwickler von Assistenten für neue Teilprozesse den Werkzeugherstellern vorbehalten. Allerdings besitzen manche Werkzeuge (z.B. das Microsoft Office-Paket oder Rational Rose) „plug-in“-Schnittstellen für die Integration von Assistenten von Drittherstellern. Für spezielle Klassen von Assistenten existieren zudem mehr oder weniger leistungsfähige Werkzeu-

ge, mit deren Hilfe sich Assistenten auf einfache Art generieren lassen. Ein Beispiel ist der InstallShield Wizard, der selbst wieder einen Assistenten für die Generierung von Installationsassistenten in Windows-Umgebungen darstellt.

- **Unterstützungsmodi**: Bei der Beurteilung der Unterstützungmodi eines Assistenten muss man zunächst zwischen verschiedenen Assistententypen unterscheiden. Passive Assistenten werden erst nach explizitem Aufruf durch den Benutzer aktiv, stellen also passive Prozessberatung dar. Die in professionellen Entwicklungsumgebungen eingesetzten Assistenten fallen in der Regel in diese Kategorie, da erfahrene Benutzer proaktive Assistenten wegen der oftmals ungenauen Ratschläge eher als störend empfinden und die Unterstützungsleistung von Assistenten lieber gezielt auf eigene Initiative hin in Anspruch nehmen. Proaktive Assistenten, die Handlungsabläufe des Benutzers beobachten und selbstständig Hilfestellung geben, finden sich in Applikationen, die in erster Linie von technisch weniger versierten Endbenutzern oder Gelegenheitsbenutzern verwendet werden (z.B. Microsoft Office). Solche Assistenten leisten aktive Prozessunterstützung. Sobald ein Assistent aktiviert ist, erzwingt er eine vordefinierte Prozessausführung. Häufig werden auch große Teile des Prozesses auf Basis der vom Benutzer gelieferten Informationen automatisiert. Der Unterstützungsmodus hängt also vom aktuellen Aktivierungszustand eines Assistenten ab.

Tab. 5 Prozessunterstützung durch Assistenten und Interface-Agenten

Art der Prozess-unterstützung	Projekt-ebene		Integrations-tiefe			Kontext-bezogenheit		Anpassbarkeit			Unterstützungsmodi			
	administrativ	technisch	extern	rechnerbasiert separat	rechnerbasiert integriert	statisch	dynamisch	fix	konfigurierbar	änderbar	passive Beratung	aktive Anleitung	erzwingend, lenkend	automatisierend
Assistenten, Interface-Agenten	o	+	–	o	+	–	+	+	o	–	o	+	o	o

2.2.4 Prozesszentrierte Umgebungen

Die bislang betrachteten Prozessunterstützungsmechanismen kommen mehr oder weniger ausgeprägt auch in solchen Entwicklungsumgebungen zum Einsatz, die nicht explizit eine prozesszentrierte Unterstützungsphilosophie verfolgen. Seit Mitte der 80er Jahre ist jedoch bei der rechnerbasierten Unterstützung von Modellierungs- und Entwurfstätigkeiten ein Trend weg von eher produktbasierten CASE-Umgebungen hin zu prozesszentrierten Umgebungen zu beobachten [FuGh94; DeKW99; FiKN94; Poh*99].

2.2.4.1 Prozessmodellierung

Die Grundidee prozesszentrierter Entwicklungsumgebungen besteht darin, dass der Integration von Prozessaspekten in eine Entwurfsumgebung eine geeignete Konzeptualisierung der zu unterstützenden Prozesse vorangehen muss. In diesem

Zusammenhang hat sich in den letzten etwa 15 Jahren mit der (Software)-Prozessmodellierung eines der aktivsten Teilgebiete innerhalb der Softwaretechnik etabliert. Das zentrales Anliegen dieses Gebiets besteht darin, Prozesse mittels Modellierung als explizite Objekte einer systematischen Erforschung und rechnergestützten Behandlung zugänglich zu machen [ABGM93; FiKN94; FuWo96; McCH95; Lonc94a; AmCF97].

Unter einem Prozessmodell versteht man eine abstrakte Repräsentation einer Familie von Prozessen unter Verwendung einer geeigneten Prozessmodellierungssprache [MDKW99]. Prozessmodelle bilden die Grundlage für die Verbreitung von Prozesswissen, Entwurf neuer Prozesse, Wiederverwendung und Anpassung existierender Prozesse, Auswahl zwischen alternativen Prozessen, Ausführung von Prozessen und die Erfassung und Analyse von Informationen über abgelaufene Prozesse [Huff96; CuKO92; CoJa99; DeKW99; FiKN94]. Diese Aktivitäten reflektieren die unterschiedliche Phasen des Lebenszyklus eines Entwicklungsprozesses und dienen drei wesentlichen Zielen, die in Publikationen über Prozessmodellierung immer wieder genannt werden [CuKO92; Dows93; Huff96; Fugg96; FiKN94; GaJa96]:

Ziele der Prozessmodellierung

- Erhöhung des Prozessverständnisses und Erleichterung der Kommunikation über Prozesse;
- Prozessverbesserung;
- Rechnerbasierte Prozessunterstützung.

Im Kontext dieser Arbeit konzentrieren wir uns auf den letztgenannten Aspekt: die Realisierung prozessmodellbasierter Unterstützungsdienste in einer Entwicklungsumgebung.

2.2.4.2 Konzeptueller Aufbau prozesszentrierter Entwicklungsumgebungen

In prozesszentrierten Entwicklungsumgebungen (PZEUen) bilden explizite Prozessmodelle die Grundlage für die anpassbare, rechnerunterstützte Steuerung des Entwicklungsprozesses. Entsprechend den Vorgaben aus den Prozessmodellen greifen so genannte *Prozessmaschinen* beratend, lenkend oder automatisierend in die Arbeitsprozesse der Entwickler ein und unterstützen den Projektmanager bei der Koordination, Kontrolle und Steuerung der Aktivitäten der einzelnen Entwickler. Für die prozessmodellgesteuerte Durchführung von Projektmanagement- und Entwicklungsaktivitäten hat sich im Englischen der Begriff *Enactment* eingebürgert, der zum einen neutral bezüglich Interpretation oder kompilierter Ausführung von Prozessmodellen ist und zum anderen zum Ausdruck bringen will, dass Prozesse von Menschen und rechnergestützten Werkzeugen gleichermaßen durchgeführt werden. Da Prozessmodelle mechanisch durch eine Prozessmaschine (üblicherweise interpretierend) ausgeführt werden, müssen Prozessmodelle in einer formalen Sprache mit einer operationalen Semantik spezifiziert werden. Die in der Softwaretechnik seit Anfang der 70-er Jahre gängigen Lebenszyklusmodelle (z.B. das Wasserfallmodell [Royc70], das Spiralmodell [Boeh88], das Fontänenmodell [HeEd93], das V-Modell [BrDr95; DrHM98]) sind hierfür ungeeignet, da sie sich zum einen auf einer extrem grobgranularen Abstraktionsebene bewegen und zum anderen nicht die für eine mechanische Ausführung erforderliche Formalität aufweisen.

Prozessmaschine

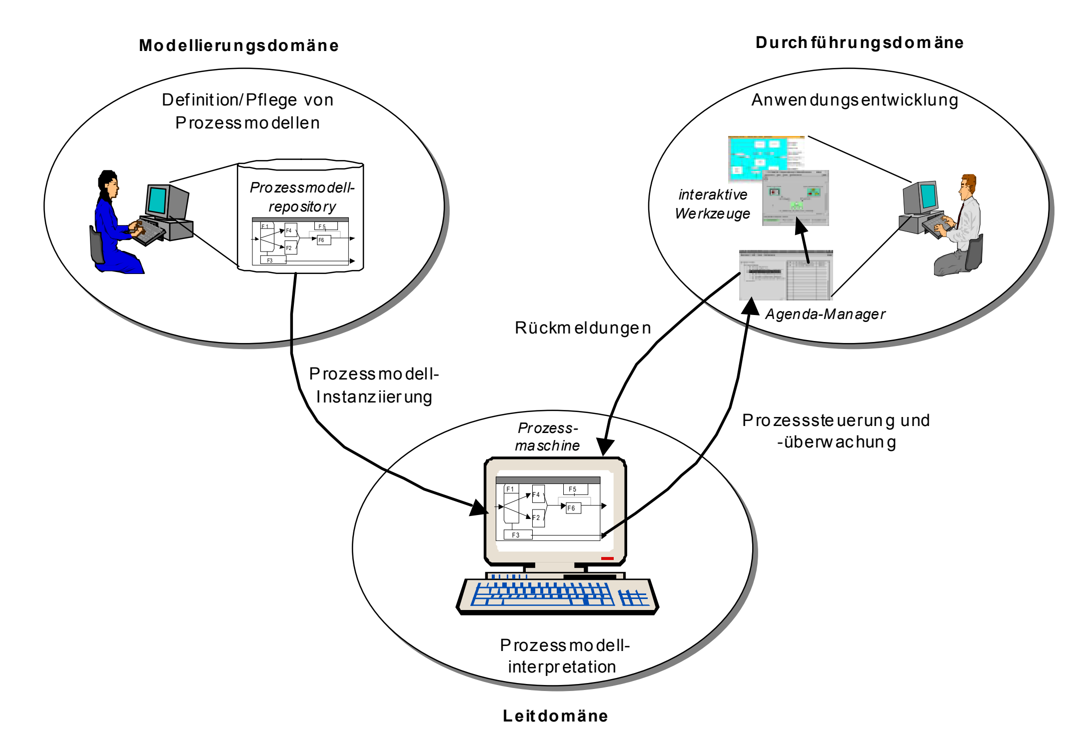


***Abb. 4**:*
Die drei Domänen der Prozessunterstützung (vgl. [DoFe94; Pohl96])

Zusammen mit den traditionellen Arbeitsplatzwerkzeugen lassen sich somit drei konzeptuelle *Aufgabenbereiche* oder *Prozessdomänen* innerhalb einer prozesszentrierten Umgebungen voneinander abgrenzen [DoFe94; Pohl96]: die *Modellierungsdomäne* (engl.: Process Modeling Domain), die *Leitdomäne* (engl.: Process Enactment Domain) und die *Durchführungsdomäne* (engl.: Process Performance Domain).

- Die *Modellierungsdomäne* umfasst alle Aufgaben, Konzepte, Notationen und Mechanismen für die Erstellung und Pflege von Prozessmodellen. Dort finden Aktivitäten wie Spezifikation, Entwurf, Implementierung, Wiederverwendung, Anpassung, Analyse, Wartung, Verwaltung und Verbesserung von Prozessmodellen statt. Neben den eigentlichen Prozessmodellen werden auch die Daten über die Prozessausführung, insbesondere die entstehenden Produkte, in der Modellierungsdomäne verwaltet.
- Die *Leitdomäne* umfasst alle Aktivitäten und Mechanismen zur Prozessunterstützung auf Basis der in Modellierungsdomäne bereitgestellten Prozessmodelle. Verkörpert wird die Leitdomäne durch eine Prozessmaschine, die die Laufzeitumgebung für instanziierte Prozessmodelle bereitstellt. Durch dynamische Fortschreibung des Ausführungszustand des Prozessmodells ist die Prozessmaschine in der Lage, in die eigentliche Prozessdurchführung unterstützend einzugreifen.
- In der *Durchführungsdomäne* führen Menschen und/oder Softwarewerkzeuge die eigentlichen Entwicklungsvorgänge durch, unabhängig davon, ob sie durch die Prozessmodellausführung angeleitet werden oder nicht.

Die von einer PZEU geleistete Prozessunterstützung lässt sich durch die typischen Beziehungen und Interaktionen der drei Prozessdomänen charakterisieren (siehe Abb. 4):

- Zur Prozessmodellausführung wird eine Kopie des Prozessmodells aus der Modellierungsdomäne in den Ausführungsmechanismus der Leitdomäne

geladen und Parameter, z.B. Entwurfsprodukte, Ressourcen und Zeitvorgaben, an projektspezifische Werte gebunden (siehe Pfeil „*Prozessmodellinstanziierung*“ in Abb. 4). Charakteristisch ist, dass bei der Initiierung der Prozessmodellausführung nicht notwendigerweise alle Prozessvariablen bereits bekannt sind, sondern mitunter erst im Laufe der Prozessausführung an konkrete Werte, beispielsweise an die Ergebnisprodukte eines vorangegangenen Arbeitsschritts, gebunden werden.

- Basierend auf der Interpretation des instanziierten Prozessmodells werden in der Leitdomäne die Aktivitäten der Durchführungsdomäne unterstützt, kontrolliert und überwacht. Dabei muss sichergestellt werden, dass die Unterstützungsvorgaben aus der Leitdomäne in der Durchführungsdomäne zumindest zur Kenntnis genommen und je nach Unterstützungsmodus (vgl. Abschnitt 2.1.5) auch befolgt werden (siehe Pfeil „*Prozesssteuerung und -überwachung*“ in Abb. 4).
- Da sich die Auswahl zwischen Handlungsalternativen und die Ergebnisse von Arbeitsschritten in der Durchführungsdomäne i.a. nicht a priori festlegen lassen, informiert die Durchführungsdomäne die Leitdomäne über den eigentlichen Prozessfortschritt (siehe Pfeil „*Rückmeldungen*“ in Abb. 4).

Die Differenzierung zwischen der Modellierungsdomäne und der Leitdomäne reflektiert den konzeptuellen Unterschied zwischen *statischem Prozessmodell* und *dynamischer Ausführung* des Prozessmodells (in Analogie zur Trennung von Programmen und Prozessen im Kontext von Betriebssystemen).

Die Trennung zwischen Leitdomäne und Durchführungsdomäne betont den Unterschied zwischen der Ausführung des Prozessmodells und der tatsächlichen Prozessausführung. Diese Unterscheidung ist fundamental, denn durch die Ausführung des Prozessmodells wird lediglich ein rechnerinternes, virtuelles Abbild des realen Prozesszustands gewartet. Der Zustand der Prozessmodellausführung und der realen Prozessausführung müssen nicht notwendigerweise übereinstimmen. In der Tat besteht ein grundlegendes Problem in PZEUen in der *Synchronisation* zwischen den Zuständen der Leit- und Durchführungsdomäne, da von der Leitdomäne auf der Basis eines unzutreffenden internen Abbilds des realen Prozesszustands kaum sinnvolle Prozessunterstützung und -kontrolle erwartet werden kann [DoFe94; Fern93a; Mont94; CDFG96; Cugo98; BaKr95]. Wir werden auf die daraus resultierenden Probleme in Abschnitt 3.3.5 noch im Detail eingehen und daraus Anforderungen an eine enge Integration zwischen den Domänen ableiten.

2.2.4.3 Schnittstellen einer PZEU

Entsprechend den im vorangegangenen Abschnitt skizzierten Funktionsbereichen lassen sich drei wesentliche externe Schnittstellen einer PZEU identifizieren, die sich an die unterschiedlichen Nutzergruppen einer PZEU richten (siehe Abb. 5):

- die Modellierungsumgebung für den Prozessmodellierer oder *Methodeningenieur*;
- die Administrationsumgebung für den Projektmanager;
- das Entwicklerfrontend für den technischen Entwickler auf der Arbeitsplatzebene.

Während der Prozessmodellierer nicht unmittelbar durch eine PZEU unterstützt wird, sondern vielmehr für die Konfiguration einer PZEU durch Bereitstellung und Pflege von Prozessmodellen zuständig ist, profitieren Projektmanager und technische Entwickler bei den auf den jeweiligen Ebenen der Systementwicklung auftretenden Aktivitäten von der Assistenzfunktion einer PZEU.

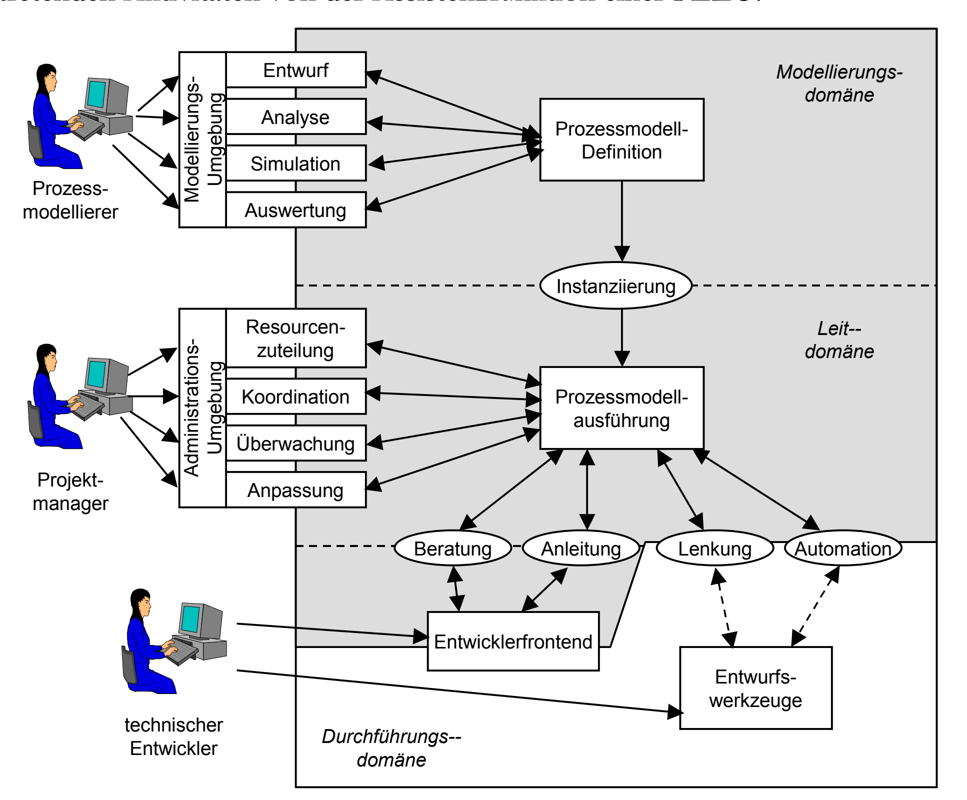


***Abb. 5**: Benutzerschnittstellen innerhalb einer PZEU*

Modellierungsumgebung

Für den Prozessmodellierer stehen in der Modellierungsdomäne eine Reihe von Werkzeugen für die Definition und Wartung von Prozessmodellen zur Verfügung (siehe Abb. 5). Dazu gehören in der Regel Editoren für den Entwurf und die Bearbeitung von Prozessmodellen und Analyse-Werkzeuge, mit deren Hilfe die erstellten Prozessmodelle auf syntaktische Korrektheit und andere statische, formale Eigenschaften (Vollständigkeit, Erreichbarkeit von Prozessschritten, Typüberprüfungen) hin überprüft werden können. Manche PZEUen bieten darüber hinaus Möglichkeiten zur Überprüfung dynamischer Aspekte von Prozessmodellen (i.a. durch Simulation) sowie des Abgleichs von Prozessmodellen mit abgespeicherten Historien bereits durchgeführter Prozesse mit dem Ziel der Aufdeckung von Schwachstellen und der Verbesserung der existierenden Prozessmodelle.

Administrations-umgebung

Der Projektmanager interagiert mit einer PZEU in der Regel über eine *Administrationsumgebung*, die der Leitdomäne zugeordnet wird (siehe Abb. 5). Zu den wesentlichen Funktionen der Administrationsschnittstelle gehören die Ressourcen- und Aufgabenzuteilung bei der Instanziierung von Prozessmodellen. Ein weiterer für das Projektmanagement wesentlicher Dienst einer PZEU ist das so genannte Prozess-Monitoring. Darunter versteht man die Überwachung von Prozessen, die unter der Kontrolle einer PZEU ablaufen, so dass der Projektmanager Prozessabweichungen erkennen kann und gegebenenfalls korrigierende Maßnahmen veranlassen kann. Außerdem kann der Projekt-Manager durch eine Gesamtsicht auf die aktuell laufenden Prozesse die Kooperation zwischen verschiedenen Entwicklern steuern und Abstimmungsprozesse veranlassen.

Entwickler-Frontend

Der technische Entwickler am Arbeitsplatz erhält Prozessunterstützung durch eine PZEU über sein *Entwickler-Frontend* (siehe Abb. 5). Basierend auf der Ausführung eines Prozessmodells und unter Berücksichtigung von prozessrelevanten Informationen aus der Durchführungsdomäne liefert die Leitdomäne der Durchführungsdomäne Unterstützung in Form von passiver Prozessberatung, aktiver Prozessanleitung, Prozesslenkung oder Prozessautomation (siehe Abschnitt 2.1.5). Für die Ausgestaltung des Entwickler-Frontends, also der Schnittstelle zwischen der Leitdomäne und der Durchführungsdomäne, wurden in PZEUen und den damit verwandten Workflow-Managementsystemen eine Reihe zum Teil sehr unterschiedlicher Interaktionsparadigmen entwickelt. Unter einem Interaktionsparadigma verstehen wir hierbei die wesentlichen Metaphern und Konzepte, mit deren Hilfe die Prozessunterstützung an den Benutzer in der Durchführungsdomäne kommuniziert wird.

- **Task- oder Agenda-Manager** stellen in den meisten PZEUen (und Workflow-Managementsystemen) die primäre Schnittstelle zwischen Leit- und Durchführungsdomäne dar (z.B. in *SPADE* [BaDF96], *HP Synervision* [DGSZ94], *Leu* [DGSZ94], *Dynamite* [HJKW96; HeKW97]). Ein Taskmanager verwaltet eine Liste von Aufgaben, die gemäß dem aktuellen Prozesszustand und der Prozessdefinition für einen bestimmten Entwickler zur Bearbeitung anstehen. Diese Schnittstelle vermittelt dem Benutzer in der Durchführungsdomäne somit eine *aktivitätsorientierte* Sicht auf den für ihn aktuell relevanten Teil des Entwicklungsprozesses.

- **Arbeitskontexte**: Im Gegensatz dazu stellen einige PZEUen die Rolle von Dokumenten oder Arbeitsordnern („Folder") stärker in den Vordergrund. Zum Beispiel basieren die PZEUen *Merlin* [JPSW94] und *ProSyt* [Cugo98] auf dem Begriff des *Arbeitskontexts*. Im Fall von Merlin werden Arbeitskontexte in der grafischen Benutzeroberfläche der PZEU als ein Netz von Dokumenten und Abhängigkeiten zwischen Dokumenten dargestellt. Jedes Dokument ist mit einem Menü assoziiert, das eine Liste von Aktionen anbietet, die auf dem ausgewählten Dokument aufgerufen werden können.

- **Visualisierung von Regelmengen**: In der regelbasierten PZEU *Marvel* [BaKa91] wird die Benutzerschnittstelle zur Leitdomäne durch eine Visualisierung von Benutzer-aktivierbaren *Regeln* realisiert. Die Aktivierung einer Regel kann in Abhängigkeit vom Zustand der Objektbank das Schalten weiterer Regeln durch Vorwärts- und Rückwärtsverkettung auslösen. Marvel ermöglichst darüber hinaus eine strukturelle Ansicht der manipulierten Artefakte, allerdings nur auf der grobgranularen Dokumentebene. Einem ähnlichen Interaktionsparadigma folgt auch die PZEU *PEACE* [ArOq94].

- **Automatischer Werkzeugaufruf**: Neben den genannten dedizierten Entwickler-Frontends (Task-Manager, Arbeitskontexte, visualisierte Regelmenge etc.) ist in den meisten PZEUen die Prozessmaschine lose mit eigentlichen Entwurfswerkzeugen der Durchführungsdomäne integriert. Allerdings beschränkt sich die Integration in den meisten Fällen auf das Starten von Werkzeugen, wobei gegebenenfalls (grobgranulare) Dokumente als Aufrufparameter übergeben werden. Während diese Art der Integration zwischen Leit- und Durchführungsdomäne bei automatischen Abläufen in Batch-artigen Werkzeugen (Compiler, Linker) durchaus angemessen und ausreichend ist, führt dies bei komplexen, interaktiven Werkzeugen zu

schwerwiegenden Problemen, die in Kapitel 3 noch genauer zu beleuchten sind.

Tab. 6 gibt einen Überblick über die Klassifizierung der Prozessunterstützung in PZEUen. Wir haben die Bewertung unterteilt, da man, wie im vorangegangenen gesehen, die Assistenzfunktion von PZEUen aus unterschiedlichen Blickwinkeln betrachten kann: aus Sicht des Projektmanagements bei der Planung und Koordination von Prozessen (vgl. auch Abschnitt 1.1.1) und aus Sicht des technischen Entwicklers, der Prozessunterstützung zum einen über PZEU-eigene Assistenzwerkzeuge wie Agenda-Manager oder Arbeitskontexte erfährt und zum anderen durch den automatischen, prozessmodellgesteuerten Aufruf von Entwurfswerkzeugen unterstützt wird.

***Tab. 6**: Prozessunterstützung in PZEUen*

Art der Prozessunterstützung	**Projekt-ebene**		**Integrations-tiefe**			**Kontext-bezogenheit**		**Anpassbar-keit**			**Unterstützungsmodi**			
	administrativ	technisch	extern	rechnerbasiert separat	rechnerbasiert integriert	statisch	dynamisch	fix	konfigurierbar	änderbar	passive Beratung	aktive Anleitung	erzwingend, lenkend	automatisierend
PZEU, Planung und Koordination	+	/	/	/	/	–	o	–	o	+	+	o	o	–
Agendamanager	/	o	–	+	–	–	o	–	o	+	+	o	o	–
Werkzeugaufruf	/	o	–	+	o	–	o	–	o	+	o	o	o	+

- **Unterstützte Projektebene**: PZEUen zielen (ebenso wie Workflow-Managementsysteme für betriebliche Abläufe) in erster Linie auf das Management von Prozessen ab, d.h. auf die Planung, Ressourcenkontrolle und Koordination der Entwicklerkooperation innerhalb eines Projekts. Die Konzentration auf die Projektmanagementebene manifestiert sich u.a. darin, dass in PZEUen Entwurfsprodukte nur auf der grobgranularen Dokumentenebene modelliert werden und lediglich größere Aufgabeneinheiten wie „Erstellung eines ER-Modells“ als atomare Prozessschritte betrachtet werden. Die im Allgemeinen grobgranulare Sicht auf den Entwicklungsprozess und die damit assoziierten Dokumente gilt auch für die Sicht des technischen Entwicklers, die ihm in den jeweiligen Entwickler-Frontends präsentiert wird. Im Gegensatz zum Projektmanager profitiert der technische Entwickler bei der systematisch-methodischen Durchführung einzelner Aufgaben somit nur eingeschränkt von der Unterstützungsleistung durch eine PZEU, indem er beispielsweise über einen Task-Manager einen Überblick über die anstehenden Aufgaben erhält oder Werkzeuge mit den benötigten Dokumenten gestartet werden. Darüber hinaus gehende feingranulare Prozessunterstützung wird von PZEUen in der Regel nicht angeboten.
- **Integrationstiefe**: PZEUen treten in der Arbeitsumgebung eines Entwicklers als zusätzliche, rechnerbasierte Assistenzwerkzeuge in Erscheinung. Wie oben beschrieben, ist die Integration mit den eigentlichen Entwurfswerkzeugen in der Regel jedoch nur rudimentär realisiert. Nach dem Start des Werkzeugs steht die Prozessausführung innerhalb des Werkzeugs nicht

mehr unter der Kontrolle der Prozessmaschine, und Rückmeldungen über die Anwendung von Werkzeugen und deren Ergebnisse müssen manuell über das Entwickler-Frontend an die Prozessmaschine übermittelt werden.

- **Kontextbezogenheit**: PZEUen sind dazu prädestiniert, kontextbezogene Prozessunterstützung liefern, da sie dynamisch ein internes Modell des aktuellen Prozesszustands pflegen. Diese Stärke relativiert sich jedoch durch die Tatsache, dass in PZEUen Prozesse in der Regel lediglich auf der für das Projektmanagement ausreichenden, relativ grobgranularen Ebene modelliert werden. Aus diesem Grund fällt die kontextbezogene Unterstützung bei der Durchführung feingranularer Aufgaben durch den technischen Entwickler relativ unpräzise aus. Außerdem spiegelt sich wegen der mangelnden Integration der Prozessmodellinterpretation mit der eigentlichen Arbeitsumgebung die im aktuellen Prozesszustand gültige Prozessanleitung nicht im Verhalten der Entwurfswerkzeuge wider.
- **Anpassbarkeit**: Eine wesentliche Stärke von PZEUen liegt in ihrer Anpassbarkeit an Prozessänderungen, da für die zu unterstützenden Prozesse explizite Prozessmodelle vorliegen. Da die Prozessmodelle in der Regel auf vergleichsweise hohem logischem Niveau ansiedelt sind und die Ausführung von Prozessmodellen in den meisten Ansätzen durch einen Prozessmodell-Interpreter erfolgt, verursachen Prozessänderungen prinzipiell einen verhältnismäßig geringen Aufwand. Die Änderungsfreundlichkeit der Prozessunterstützung hängt allerdings auch davon ab, ob der zugrunde liegenden Prozessmodellierungsformalismus auch von Programmiersprachen gewünschte Eigenschaften wie Modularität, Kapselung und Wiederverwendung unterstützt.
- **Unterstützungsmodi**: Existierende PZEUen bzw. die ihnen zugrunde liegenden Prozessmodellierungsansätze tendieren dazu, Entwicklungsprozesse möglichst vollständig zu erfassen und präskriptiv zu unterstützen. Ein wesentliches Motiv für diese Fokussierung auf lenkende und automatisierende Unterstützungsmodi ist in den Schwierigkeiten zu sehen, die sich bei der Handhabung von Abweichungen zwischen der tatsächlichen und der im Prozessmodell intendierten Prozessdurchführung ergeben. Im Gegensatz zur Prozesslenkung und -automation kann bei der passiven Prozessberatung und der aktiven Prozessanleitung die Prozessmaschine nicht davon ausgehen, dass die intendierte Prozessausführung vom Entwickler auch tatsächlich befolgt wird. Der aktuelle Prozesszustand kann daher nicht aus der internen Fortschreibung der Prozessmodellausführung abgeleitet werden, sondern muss ständig mit Zustandsinformationen aus der Durchführungsdomäne abgeglichen werden.

2.3 Fazit

In diesem Kapitel haben wir zunächst ein Klassifikationsschema entwickelt, das eine Bewertung prozessorientierter Unterstützungsfunktionen hinsichtlich der Merkmale *unterstützte Projektebene*, *Integrationstiefe*, *Kontextbezogenheit*, *Anpassbarkeit* und Abdeckung des Spektrums unterschiedlicher *Unterstützungsmodi* ermöglicht. Anhand dieses Klassifikationsschemas haben wir die Unterstützungsleistung von Methodenhandbüchern, Hilfesystemen, Assistenten und Interface-Agenten sowie

prozesszentrierten Umgebungen diskutiert. Im Folgenden stellen wir die Stärken und Schwächen der untersuchten Ansätze noch einmal zusammenfassend gegenüber (Abschnitt 2.3.1) und definieren darauf aufbauend die wesentlichen Charakteristika *prozessintegrierter Werkzeuge* (Abschnitt 2.3.2), welche die Vorteile existierender Ansätze vereinigen sollen.

2.3.1 Vergleich der Prozessunterstützungsansätze

Einen nur geringen Beitrag zur Prozessunterstützung können Methoden- und Projekthandbücher leisten, da das in ihnen dokumentierte Prozesswissen statisch ist und nicht innerhalb der rechnerbasierten Umgebung kontextsensitiv abgerufen werden kann. Änderungen der in einem Handbuch fixierten Prozessdefinitionen ziehen einen hohen Aufwand nach sich.

Dagegen besteht die wesentliche Stärke prozesszentrierter Umgebungen in ihrer gerade in kreativen Entwurfsdomänen erforderlichen Flexibilität hinsichtlich der Definition neuer Prozesse oder der Anpassung von existierenden Prozessen, da die von ihnen geleistete Prozessunterstützung auf expliziten und austauschbaren Prozessmodellen basiert. Aufgrund der Ausführbarkeit der Modelle durch Prozessmaschinen sind prozesszentrierte Umgebungen in der Lage, ein internes Modell des aktuellen Prozesszustands in der Leitdomäne dynamisch fortzuschreiben und somit die Durchführungsdomäne kontextsensitiv zu unterstützen. Von der Assistenzfunktion prozesszentrierter Umgebungen profitiert in existierenden Ansätzen allerdings primär das Projektmanagement, da dort die Unterstützung administrativer Vorgänge wie Aufgabendekomposition und -verteilung, Verwaltung von Ressourcen, Koordination von Entwicklern, Prozessüberwachung und Fortschrittskontrolle im Vordergrund steht.

Bei der methodischen Anleitung feingranularer Entwickleraktivitäten weisen prozesszentrierte Umgebung jedoch gravierende Schwächen auf, die in erster Linie auf die nur *lose Integration* der Modellierung- und Leitdomäne mit der Durchführungsdomäne, d.h. den Werkzeugen, zurückzuführen sind. Die Kommunikation mit dem Benutzer erfolgt meist über separate Assistenzwerkzeuge, die eine bestimmte Sicht auf den Entwicklungsprozess bieten und manuell vom Benutzer anzugebende Rückmeldungen aus der Durchführungsdomäne entgegennehmen. Die Integrationstiefe mit den Entwicklungswerkzeugen ist gering und beschränkt sich in der Regel auf den grobgranularen Aufruf von Werkzeugen, wodurch gewisse Abläufe automatisiert werden können. Die schwierige Kontrolle des Einsatzes und der Verwendung von Werkzeugen durch den Benutzer führt dazu, dass Prozesslenkung (*process enforcement*), d.h. die prozesskonforme Durchführung von Entwicklungsaktivitäten, in prozesszentrierten Umgebungen nicht sichergestellt werden kann.

Dedizierte, werkzeugorientierte Unterstützungssysteme wie Hilfesysteme, Assistenten und Interface-Agenten bieten zwar eine hohe Unterstützungsqualität für den Benutzer einer Entwicklungsumgebung, da sie unmittelbar mit den Werkzeugen, mit denen der Entwickler den eigentlichen Prozess durchführt, interagieren. Allerdings verkörpern sie Prozesswissen nur in indirekter und hartkodierter Form. Zudem sind sie wegen der engen Verschränkung mit dem jeweiligen zugrunde liegenden Werkzeug nicht für die Prozessunterstützung werkzeugübergreifender Abläufe ausgelegt.

Tab. 7: Prozessunterstützungsansätze im Vergleich

Tab. 7 fasst die wesentlichen Merkmale der Prozessunterstützung durch Handbücher, Hilfesysteme, Assistenten und prozesszentrierte Entwicklungsumgebungen vergleichend zusammen.

Art der Prozessunterstützung	**Projektebene**		**Integrationstiefe**			**Kontextbezogenheit**		**Anpassbarkeit**			**Unterstützungsmodi**			
	administrativ	technisch	extern	rechnerbasiert separat	rechnerbasiert integriert	statisch	dynamisch	fix	konfigurierbar	änderbar	passive Beratung	aktive Anleitung	erzwingend, lenkend	automatisierend
Methoden- und Projekthandbücher	+	+	+	–	–	+	–	+	–	–	+	–	–	–
Hilfesysteme statisch, uniform	o	+	–	+	–	+	–	+	o	–	/	/	–	–
dynamisch, individuell	o	+	–	+	o	–	+	+	o	–	/	/	–	–
passiv	o	+	–	+	–	/	/	+	o	–	+	–	–	–
aktiv	o	+	–	+	o	–	+	+	o	–	+	o	–	–
Assistenten, Interface-Agenten	o	+	–	o	+	–	+	+	o	–	o	+	o	o
PZEU, Planung und Koordination	+	/	/	/	/	–	o	–	o	+	+	o	o	–
Agendamanager	/	o	–	+	–	–	o	–	o	+	+	o	o	–
Werkzeugaufruf	/	o	–	+	o	–	o	–	o	+	o	o	o	+

2.3.2 Einordnung prozessintegrierter Werkzeuge

Die Hauptthese dieser Arbeit lautet, dass durch eine engere Integration der Modellierungs- und Leitdomäne (Prozessmodelle und Prozessmaschine) mit der Durchführungsdomäne (Entwicklungswerkzeuge) die Vorteile der flexiblen Anpassbarkeit prozesszentrierter Umgebungen einerseits und die hohe Unterstützungsqualität werkzeugbezogener Assistenten andererseits kombiniert werden können. Werkzeuge, die ihre Arbeitsweise den in der Modellierungsdomäne definierten und in der Leitdomäne ausgeführten Prozessmodellen unterordnen, bezeichnen wir als *prozessintegrierte Werkzeuge*.

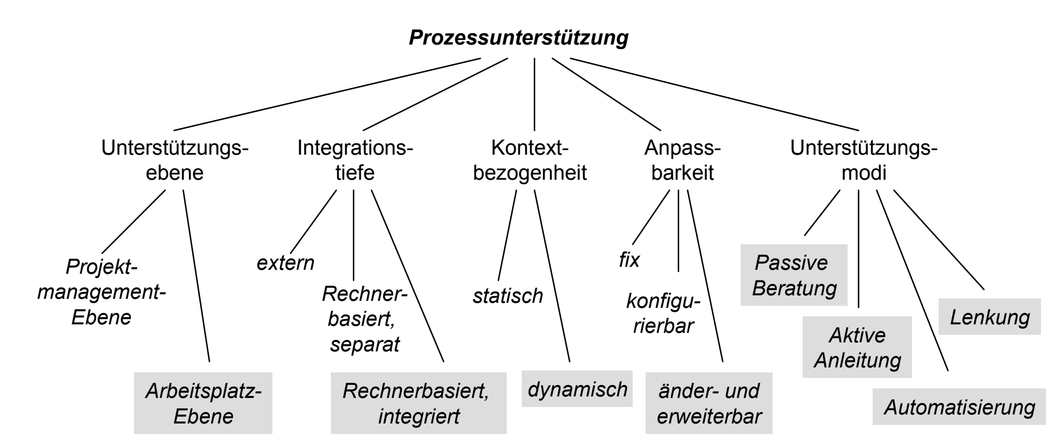
Anhand unserer Klassifikationskriterien können wir die wesentlichen Merkmale einer Unterstützung durch prozessintegrierte Werkzeuge charakterisieren:

- **Fokussierung auf Arbeitsplatzebene**: Die in den Werkzeugen stattfindenden Abläufe werden modellierungsmäßig auf *feingranularer* Ebene betrachtet. Dadurch ist es uns möglich, anders als bei den meisten Ansätzen aus dem Bereich Prozesszentrierte Entwicklungsumgebungen oder Workflow-Managementsysteme die Arbeitsplatzebene in das Zentrum der Unterstützung zu stellen. Prinzipiell lassen sich natürlich auch Arbeitsabläufe auf der Projektmanagementebene durch prozessintegrierte Werkzeuge unterstützen; dies ist jedoch nicht Gegenstand dieser Arbeit.

- **Rechnerbasierte, integrierte Unterstützung**: Die Prozessunterstützung, wie sie durch Prozessmodelle und Prozessmaschine verkörpert wird, ist unmittelbar in der Werkzeugumgebung sichtbar, und der Benutzer benötigt keine zusätzlichen Benutzerschnittstellen, um mit der Prozessmaschine zu interagieren.
- **Änderbarkeit und Erweiterbarkeit**: Prozessintegrierte Werkzeuge verkörpern kein eigenes, hartkodiertes Prozesswissen; dieses ist in der Modellierungsdomäne in Form explizit modellierter Prozessfragmente offengelegt. Daher können die zu unterstützenden Abläufe mit vergleichsweise geringem Aufwand an organisations- und projektspezifische Bedürfnisse angepasst werden und sind mit zunehmendem Prozesswissen auch erweiterbar.
- **Dynamische Prozessunterstützung**: Der aktuelle Prozesszustand in der Durchführungsdomäne lässt sich am besten durch den Zustand der Werkzeuge (geladene Dokumente) und dort stattfindende Ereignisse (Auswahl von Produkten und Kommandos, Durchführung von Aktionen) charakterisieren. Prozessintegrierte Werkzeuge gleichen ihren Zustand ständig mit der Prozessmodellausführung in der Leitdomäne ab. Dadurch wird gewährleistet, dass die angebotene Prozessunterstützung stets auf den aktuellen Kontext zugeschnitten ist.
- **Unterschiedliche Unterstützungsmodi**: Die direkte Rückwirkung der Prozessmodellinterpretation auf das Werkzeugverhalten hat mehrere Facetten, die in unterschiedlichen Unterstützungsmodi resultieren:
  - *Feingranulare Kontrolle (Automation)*: es stehen Werkzeugschnittstellen zur Verfügung, über die die Prozessmaschine Aktionen in laufenden Werkzeugen anstoßen kann. Dadurch können in interaktiven Werkzeuge gut verstandene Abläufe auch über Werkzeuggrenzen hinweg automatisiert werden;
  - *Prozesssensitive Einschränkung der Interaktionsmöglichkeiten (Prozesslenkung):* Prozessintegrierte Werkzeuge sind in der Lage, die Interaktionsmöglichkeiten des Benutzers dynamisch auf die aktuelle Prozesssituation anzupassen, indem sie den Zugriff auf Werkzeugfunktionen und Produkte gemäß den Vorgaben aus der Leitdomäne einschränken. Dadurch lassen sich Abläufe innerhalb eines vom Prozessmodell vorgegebenen „Korridors" lenken, ohne die Flexibilität des Benutzers vollständig einzuschränken.
  - *Unterstützung bei der Aktivierung von Prozessfragmenten (passive Prozessberatung und aktive Prozessanleitung)*: Prozessintegrierte Werkzeuge unterstützen den Entwickler beim Abgleich der aktuell vorliegenden Prozesssituation mit den Prozessdefinition und ermöglichen so (auf Anfrage des Benutzers oder selbständig) die Aktivierung von Prozessfragmenten. Der aktuelle Werkzeugzustand (selektierte Produkte, aktivierte Kommandos) definiert dabei den Kontext für die Anwendbarkeit von Prozessfragmenten. Aus Benutzersicht besteht somit kein Unterschied in der Aktivierung einer elementaren Werkzeugfunktion und eines Prozessfragments.

Abb. 6 illustriert die Einordnung prozessintegrierter Werkzeuge in den durch die unterschiedlichen Kriterien aufgespannten Raum möglicher Prozessunterstützungsfunktionen.

***Abb. 6**:*
Einordnung prozessintegrierter Werkzeuge



Kapitel 3

Integrationsansätze

Im vorangegangenen Kapitel haben wir argumentiert, dass eine Prozessintegration von Werkzeugen, die auf einer stärkeren Berücksichtigung von Werkzeugen in prozesszentrierten Umgebungen beruht, zu einer Unterstützung führt, die direkt in der Arbeitsumgebung des technischen Entwicklers sichtbar ist, kontextsensitiv in den Arbeitsprozess eingreift, das gesamte Spektrum möglicher Unterstützungsmodi abdeckt und darüber hinaus einfach anpassbar ist.

Ziel dieses Kapitels ist es, die Anforderungen an die dafür erforderliche Integration zwischen der Modellierungs- und Leitdomäne einerseits und der Durchführungsdomäne andererseits genauer herauszuarbeiten und mit existierenden Integrationsstrategien und -mechanismen aus der Literatur in Beziehung zu setzen. Diese Anforderungen kennzeichnen den Übergang von prozesszentrierten Umgebungen, denen lediglich die explizite Modellierung von Prozessen zugrunde liegt, hin zu *prozessintegrierten* Umgebungen, in denen auch die Werkzeuge der Durchführungsdomäne eine prozessmodellkonforme Arbeitsweise aktiv unterstützen.

Der Rest des Kapitels ist wie folgt gegliedert. In Abschnitt 3.1 geben wir zunächst einen kurzen Überblick über verschiedene Klassifikationsmodelle, die in der Literatur zur konzeptionellen Strukturierung des Integrationsproblems vorgeschlagen worden sind. In Abschnitt 3.2 diskutieren wir die grundsätzlichen Möglichkeiten und Grenzen der a posteriori- und a priori-Integration und motivieren, warum wir in dieser Arbeit den Schwerpunkt auf letztere legen. Die spezifischen Anforderungen an die Integration der Prozessdomänen werden Abschnitt 3.3 zusammen mit möglichen Lösungsansätzen ausführlich dargestellt. Abschnitt 3.4 fasst die Ergebnisse des Kapitels zusammen.

3.1 Perspektiven der Werkzeugintegration

Die Prozessintegration von Werkzeugen ist keine völlig neue Fragestellung, sondern führt unterschiedliche Integrationsprobleme zusammen, für die mittlerweile jeweils zum Teil recht ausgereifte Lösungsansätze in der Literatur vorliegen. In der Tat gehört die Werkzeugintegration zu den am intensivsten untersuchten Teilgebieten der Forschung über Entwurfsumgebungen und wird von manchen Autoren aus dem Bereich der Softwaretechnik gar als der „heilige Gral der SEE- und CASE-Technologie“ bezeichnet [BrEM92]. Nichtsdestotrotz hat sich, auch nachdem vor nunmehr 20 Jahren mit dem „Stoneman-Report“ [DoD#80] der Startschuss für eine systematische Beschäftigung mit integrierten Entwurfsumgebungen gefallen ist, das Integrationsproblem einfachen Lösungen standhaft widersetzt und gilt

immer noch als eine schwierige Herausforderung für Forscher und Praktiker aus diesem Bereich [Meye91; ScBr93; Bro*94; Kelt93].

Was bedeutet Integration?

Insbesondere hat sich die Frage, welche Schlüsseleigenschaften eine integrierte Entwurfsumgebung kennzeichnen, als erstaunlich schwierig herausgestellt [Wass90; ThNe92; Sche93; Bro*94]. Ganz generell verbindet man mit Integration in einer Entwurfsumgebung den Wunsch nach besserer Koordination und stärkerer Vereinheitlichung von zunächst unabhängigen Einzelkomponenten. Insbesondere steht hinter dem Integrationsbegriff die Erwartung, dass eine integrierte Entwurfsumgebung ihren Verwendern, d.h. den Entwicklern, effizientere Arbeitsabläufe ermöglicht und somit einen höheren Nutzwert bietet als die Summe ihrer Einzelkomponenten [ThNe92; Wass90; Sche93; Kelt93].

Um die Integriertheit und Integrierbarkeit in Entwurfsumgebung besser untersuchen zu können, sind in der Literatur eine Reihe von Klassifikationsmodellen vorgeschlagen worden. Diese liefern einen konzeptuellen Rahmen, in dem verschiedene Aspekte des Integrationsproblems einer getrennten Betrachtung unterzogen werden können, aber auch Querbezüge zwischen einzelnen Integrationsaspekten untersucht werden können. Im Folgenden diskutieren wir kurz die wichtigsten dieser zum Teil aufeinander aufbauenden Klassifikationsmodelle.

3.1.1 Integration als Informationsmanagement

Eine Sichtweise, die die Architektur vieler Entwurfsumgebungen insbesondere aus dem akademischen Bereich geprägt hat, betrachtet das Integrationsproblem in erster Linie als eine Frage des *Informationsmanagements* [ScBr93]. Kern der dahinterstehenden Vision einer integrierten Umgebung ist die Forderung, dass die Entwurfsumgebung dem Entwickler bei seinen alltäglichen Tätigkeiten einen unmittelbaren und unkomplizierten Zugriff auf jede benötigte Information ermöglichen soll. Die effiziente und situative Bereitstellung der richtigen Entwurfsobjekte zum richtigen Zeitpunkt wird als Schlüsselfaktor zur Steigerung der Produktivität angesehen. Häufig werden Hypertext-Metaphern, d.h. das Navigieren und Stöbern („Browsing") entlang miteinander in Beziehung stehender Informationseinheiten, verwendet, um die Arbeitsweise in solchen Umgebungen zu charakterisieren.

Repository-zentrierte Sichtweise auf Integration

Auf Architekturebene wird der Integrationsgedanke durch ein so genanntes *Repository* verkörpert, welches das Rückgrat der Entwurfsumgebung bildet. In dem Repository werden alle anfallenden Entwurfsobjekte erfasst, strukturiert und zueinander in Beziehung gesetzt, seien es Anforderungsmodelle, Entwurfsspezifikationen, Software-Bausteine, Testpläne oder auch „weiche" Informationseinheiten wie Besprechungsprotokolle oder Entwurfsentscheidungen. Das Repository ermöglicht es den Werkzeugen, Informationen in Form von Entwurfsergebnissen auszutauschen. Die Suche nach geeigneten Strukturierungskonzepten, d.h. Datenmodellen, für Entwurfsdaten, deren effiziente technische Verwaltung durch „Non-Standard"-Datenbanktechnologie sowie das Änderungsmanagement in Repositories haben die Forschung in diesem Bereich dominiert [Bro*94; BeDa94; Ber*99; Ortn99; NiJa99; WaJo93].

3.1.2 Integration als eine Menge von orthogonalen Dimensionen

Wasserman's Integrationsdimensionen [Wass90]

Von Wasserman stammt die wohl einflussreichste Publikation zur Klassifikation des Integrationsproblems [Wass90]. Die zentrale Idee in Wasserman's Vorschlag besteht darin, Entwurfsumgebungen bezüglich ihrer Integrationsmechanismen entlang mehrerer, zueinander orthogonaler Dimensionen zu bewerten. Er erweitert die rein Repository-zentrierte Perspektive, indem er neben datenbezogenen Aspekten (*Datenintegration*) auch die Überbrückung von Hardware-, Betriebssystem- und Netzwerkheterogenität (*Plattformintegration*), die Vereinheitlichung der Benutzeroberfläche (*Präsentationsintegration*), die Steuerbarkeit von Werkzeugen (*Kontrollintegration*) und die Einbettung von Werkzeugen in einen definierten Arbeitsprozess (*Prozessintegration*) in seine Analyse des Integrationsproblems einbezieht.

Fokussierung auf Integrations-mechanismen

Ein wichtiges Merkmal in Wasserman's Klassifikationsmodell ist die Fokussierung auf *Integrationsmechanismen*, d.h. die Qualität und Tiefe der Integration wird am Vorhandensein bestimmter Integrationsdienste festgemacht. Hierbei konzentriert er sich insbesondere auf die Aspekte Daten-, Kontroll- und Präsentationsintegration, für die er eine Skalierung der entsprechenden drei Integrationsachsen einführt. So wird beispielsweise die Datenintegrationsachse danach unterteilt, ob in einer Entwurfsumgebung das Dateisystem, eine Datenbank oder ein dediziertes Objektmanagementsystem zum Austausch von Daten zwischen Werkzeugen verwendet wird. Die Position eines Integrationsmechanismus auf der jeweiligen Achse suggeriert zumindest implizit seine Güte. Je nach den vorhandenen Integrationsmechanismen lassen sich so unterschiedliche Entwurfsumgebungen innerhalb des dreidimensionalen Raums einordnen. Eine solche Einordnung ist für den Vergleich verschiedener Produkte zwar hilfreich, hat aber wegen der groben Einteilung nur Übersichtscharakter.

3.1.3 Integration im Spannungsfeld von Basismechanismen und Prozessen

Die von Wasserman vorgenommene Separierung der einzelnen Integrationsaspekte wurde von den meisten Autoren als sehr hilfreich empfunden und gilt heute als allgemein akzeptiert. Kritik entzündete sich allerdings daran, dass Wasserman bereits die Präsenz bestimmter Integrations*mechanismen* mit Integration gleichsetzt, ohne auf *semantische* Aspekte einzugehen [Bro*94; ThNe92].

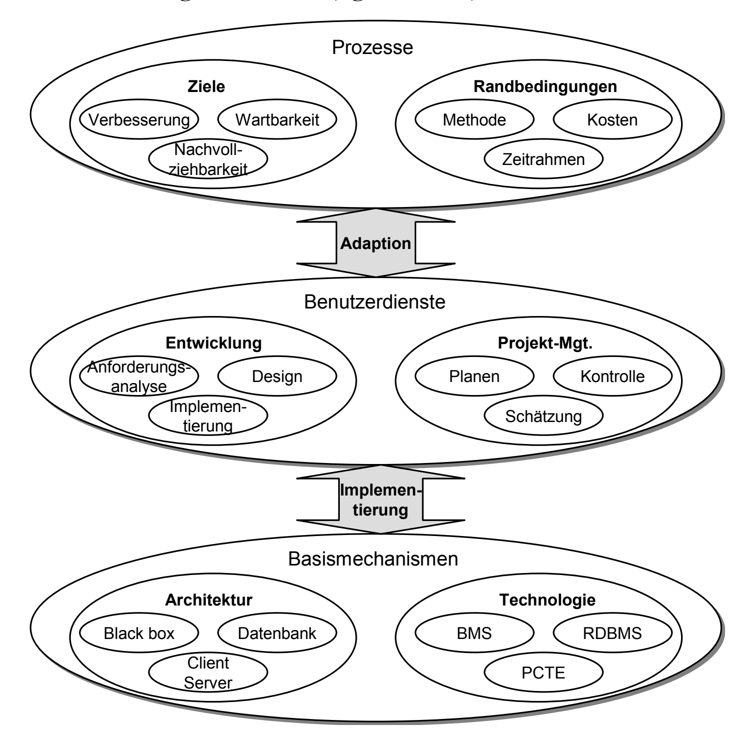
Thomas und Nejmeh setzen in [ThNe92] zwar auf Wasserman's Klassifikationsmodell auf, machen aber Integration nicht mehr am Vorhandensein bestimmter Integrationsmechanismen fest, sondern fassen sie als Charakteristikum der Beziehung zwischen zwei oder mehreren Komponenten einer Entwurfsumgebung auf. Sie identifizieren mehrere Arten von Beziehungen zwischen Komponenten, die für Integration von Belang sind: Beziehungen zwischen Werkzeugen; Beziehungen zwischen Werkzeugen und Rahmenwerksdiensten; Beziehungen zwischen Werkzeugen und Prozessen.

Unterscheidung zwischen semantischen und mechanistischen Integrationsaspekten

In eine ähnliche Richtung gehen die Überlegungen von Brown et al., die in [BrFe92; Bro*94] eine Analysetechnik für die Integration in Entwurfsumgebungen vorschlagen, die, stärker noch als das Klassifikationsmodell von Thomas und

Nejmeh, auf die Unterscheidung zwischen *semantischen* und *mechanistischen* Aspekten abzielt. Dazu betrachteten sie die drei Ebenen der *Basismechanismen*, der *Benutzerdienste* und der *Prozesse*. Sie argumentieren, dass auf jeder dieser Ebenen und zwischen den Ebenen jeweils spezifische Integrationsfragestellungen auftreten, die eine getrennte Behandlung erfordern (vgl. Abb. 7).

***Abb. 7**: Integration im Spannungsfeld zwischen Prozessen und Basismechanismen [Bro*94]*



Basismechanismen

Die Ebene der *Basismechanismen* befasst sich mit den prinzipiellen Architekturentscheidungen, die einer integrierten Umgebung zugrunde liegen, und den für die Realisierung verwendeten Rahmenwerksdiensten. Hier wird also in erster Linie die Integrationsfragestellung, *wie*, d.h. mit welchen *Mechanismen*, in einer Entwurfsumgebung die einzelnen Komponenten miteinander verbunden werden, untersucht.

Benutzerdienste

Die Ebene der *Benutzerdienste* korrespondiert mit einer abstrakten Beschreibung der Funktionalität der Umgebung. Integration auf dieser Ebene beschäftigt sich mit der Frage, welche Dienste aus Sicht des Benutzers von der Umgebung angeboten werden und in welcher Beziehung diese zueinander stehen. Auf dieser Ebene wird die *Semantik* der unterschiedlichen Integrationsaspekte (insbesondere Daten-, Kontroll- und Präsentationsintegration) genauer geklärt.

Prozesse

Die *Prozessebene* befasst sich mit den Zielen und Einschränkungen, die von den zu unterstützenden Entwurfsprozessen herrühren und somit einen Kontext für die Verwendung und Integration der Benutzerdienste aufspannen.

Realisierung

Zwischen den Ebenen existieren im Wesentlichen zwei *Wechselwirkungen*. Die Beziehung zwischen Ebene der Benutzerdienste und der Basismechanismen wird als *Realisierungsbeziehung* verstanden. Hierbei gibt es selbstverständlich keine eindeutige Zuordnung zwischen Elementen der beiden Ebenen. Benutzerdienste werden unter Verwendung unterschiedlicher Basisdienste implementiert, während umgekehrt ein Basisdienst bei der Realisierung mehrerer Benutzerdienste verwendet kann.

Adaption

Die Wechselwirkung zwischen der Prozess- und der Benutzerdienstebene wird als *Adaptionsbeziehung* aufgefasst. Die zu unterstützenden Prozesse definieren Richt-

linien und Einschränkungen für die Verwendung von Benutzerdiensten und legen eine Reihenfolge in der Verwendung der Dienste fest. Indem entlang eines vorgegebenen Entwurfsprozesses bestimmte Dienste die Eingabe für nachfolgende Dienste liefern, lässt sich identifizieren, welche Dienste miteinander interagieren müssen und welche Schnittstelle sie aufweisen müssen. Der Wert dieser Betrachtungsweise liegt darin, dass die Analyse der zu unterstützenden Entwurfsprozesse dem Umgebungsintegrator Aufschluss darüber gibt, auf welche Integrationsbeziehungen zwischen Werkzeugen er sich konzentrieren sollte, anstatt jedes Werkzeug vollständig mit jedem anderen zu integrieren.

3.1.4 Fazit

Werkzeugintegration kann aus unterschiedlichen Blickwinkeln betrachtet werden. Während frühe Ansätze Integration hauptsächlich als eine Frage des koordinierten Informationsmanagements zwischen Werkzeugen sehen, hat sich seit dem Klassifikationsschema von Wasserman [Wass90] eine reichhaltigere Sichtweise auf das Integrationsproblem durchgesetzt, welche insbesondere auch den Begriff der Prozessintegration umfasst.

Wasserman verknüpft Prozessintegration mit dem Vorhandensein expliziter Prozessmodelle und einer Prozessmaschine, also Elementen der Modellierungs- bzw. Leitdomäne, die mit den Entwicklungswerkzeugen, also der Durchführungsdomäne, selbst wieder über Mechanismen der übrigen Integrationsdimensionen verbunden sind. Prozessintegration ist also für Wasserman nichts anderes als die Erweiterung einer traditionellen (produktorientierten) Entwicklungsumgebung zu einer prozesszentrierten Umgebung. Darüber hinaus gibt er keine konkreten Hinweise, welche spezifischen Anforderung an die zu integrierenden Entwicklungswerkzeuge bzw. an die zugrunde liegenden Integrationsmechanismen zu stellen sind.

Ähnlich vage bleiben Thomas und Nejmeh [ThNe92], die Prozessintegration dahingehend definieren, dass „Werkzeuge effektiv zur Unterstützung eines definierten Prozesses interagieren“ sollen, und zwar hinsichtlich der Prozessschritte, -ereignisse und -einschränkungen, die sich aus einer Definition des Prozesses ergeben.

In der Terminologie von [Bro*94] steuert, kontrolliert und adaptiert die Prozessebene (Modellierungs- und Leitdomäne) die Dienste der Benutzerebene (Durchführungsdomäne) unter Verwendung von Basismechanismen und Integrationstechnologien, die sich wieder den Dimensionen Daten-, Kontroll- und Präsentationsintegration zuordnen lassen. Wichtig erscheint uns hier, dass Brown et al. den Charakter von Prozessintegration als einem *Adaptabilitätsproblem* hervorheben.

Es reicht also nicht aus, Werkzeuge in einen einmal definierten Prozess einzubinden. Dies könnte auch durch hartkodierte Integrationsbeziehungen erreicht werden. Vielmehr muss bei der Auswahl und Ausgestaltung von Mechanismen zur Werkzeugintegration berücksichtig werden, dass Prozesse hochgradig dynamisch sind. Die daraus resultierende Adaptabilität bezieht sich dabei auf zwei unterschiedliche Dynamikaspekte. Zum einen müssen sich Änderungen des *Ausführungszustands* definierter Prozesse (*process enactment*) unmittelbar in einem angepassten Verhalten der Werkzeuge widerspiegeln. Diese Art von Anpassbarkeit nennen

Ausführungs- und Modellierungs-adaptabilität

wir *Ausführungsadaptabilität*. Zum anderen müssen Änderungen an den Prozessmodellen selbst (*process evolution*) auf die Ebene der Benutzerdienste propagiert werden. Hier sprechen wir von *Modellierungsadaptabilität*. Es sei noch angemerkt, dass die Grenze zwischen Adaptabilität zur Modellierungs- und zur Ausführungszeit zunehmend verschwimmt; aktuelle Forschungsansätze widmen sich verstärkt dem schwierigen Problem der Evolution *laufender* Prozesse [BaFG93; BoTa96; Bey*00; GrRW00; CNWL00].

Konkrete Anforderungen und Hinweise, wie die Prozessintegration von Modellierungs- und Leitdomäne mit der Durchführungsdomäne in einem *ganzheitlichen Ansatz* auf Basis der Integrationsmechanismen der unteren Ebene auszugestalten ist, fehlen jedoch bei den oben genannten Arbeiten ebenso wie in anderen Publikationen zu diesem Thema [ChNo92; FeOh91; DoFe94; GiKa91] und gelten allgemein noch als wenig verstanden [Bro*94; JaHu98; SJHB96; Böhm98; BeMü99; EmFi96]. Auch Rahmenmodelle für integrierte Entwurfsumgebungen [Kelt93] wie das so genannte „Toaster"-Modell der europäischen ECMA[2] [ECMA93], das „Prisma"-Modell der amerikanischen NIST[3] [NIST93] oder das CAD-Referenzmodell [Abel95] listen in einem Katalog von Basisdiensten zwar Dienste für das Prozessmanagement (also im wesentlichen Prozessmaschinen und Prozessmodelle) auf, ohne jedoch genaueren Aufschluss über das Zusammenspiel dieser Dienste mit den anderen Basisdiensten und den zu integrierenden Werkzeugen zu geben.

3.2 Integrationsvoraussetzungen

Vor einer Diskussion der unterschiedlichen Integrationsanforderungen ist es zunächst wichtig, sich über die äußeren Randbedingungen für die Einbindung eines Werkzeugs in eine prozessintegrierte Umgebung Klarheit zu verschaffen. Hier lassen sich im Wesentlichen drei verschiedene Integrationsszenarien unterscheiden, die einen wesentlichen Einfluss darauf haben, in welcher Granularität und Tiefe die Integration eines Werkzeugs überhaupt gelingen kann [VaKa96; RaSt92; BeMü99; JaBu96; EmFi96]:

- **Blackbox-Integration**: Bei dieser schwächsten Form der Integration sind Werkzeuge lediglich als ausführbare Programme (*binaries*) verfügbar. Die Einflussnahme von außen beschränkt sich auf das Starten und Beenden des Werkzeugs, wobei gegebenenfalls über die Kommandozeile oder über Umgebungsvariablen Startparameter (Eingabedaten, spezifische Funktionen) übergeben und nach der Werkzeugausführung Ausgabedaten und Status-Codes, die Aufschluss über den Erfolg der Werkzeugdurchführung geben, entgegengenommen werden können. Blackbox-Integration eignet sich nur für Batch-artig arbeitende Werkzeuge wie Compiler, Linker etc., deren Aktivierungsdauer eineindeutig einem atomaren Prozessschritt entspricht [Böhm98]. Bei interaktiven Werkzeugen (z.B. Editoren, Browser, Analysewerkzeuge) stellt ein wiederholtes Starten eines Werkzeugs bei jeder Anforderung einer Werkzeugfunktion durch die Prozessmaschine in der Regel

[2] ECMA: European Computer Manufacturers Association

[3] NIST: National Institute of Standards and Technology

eine schlechte Lösung dar. Dies liegt zum einen an den in der Regel *nicht vernachlässigbaren Aufstartzeiten* interaktiver Werkzeuge und zum anderen am *inkrementellen Interaktionsstil* in solchen Werkzeugen, also der sukzessiven Weiterbearbeitung eines einmal geladenen Dokuments und seines komplexen internen Zustands durch den Benutzer. Überdauert jedoch ein interaktives Werkzeug die Dauer des Prozessschritts, für dessen Unterstützung das Werkzeug von der Prozessmaschine aufgerufen wurde, verliert die Leitdomäne die Kontrolle über die Benutzeraktivitäten in diesem Werkzeugen und kann eine definitionskonforme Prozessdurchführung nicht mehr sicherstellen ([Böhm98; EmFi96; AnGr94], siehe auch Abschnitt 3.3.5).

- **Greybox-Integration**: Hier steht der Quellkode des Werkzeugs für Anpassungen an die speziellen Anforderungen einer prozesszentrierten Umgebung ebenfalls nicht zur Verfügung. Anders als bei der Blackbox-Integration stellt das Werkzeug jedoch eine eigene Sprache für Erweiterungen (z.B. *E-Lisp* für den Texteditor *emacs* [Stal81]) oder programmierbare Laufzeitschnittstellen (*Application Programming Interfaces*, APIs) bereit, über die das Werkzeug mit der Leitdomäne interagieren kann. Dadurch kann ein Werkzeug auch während seiner Laufzeit mit der Prozessmaschine Daten und Kontrollinformationen austauschen, muss also nicht für jeden Prozessschritt erneut gestartet werden und erlaubt so die Beibehaltung eines inkrementellen Interaktionsstils. Insbesondere können so gezielt individuelle, feingranulare Dienste eines Werkzeugs von der Prozessmaschine direkt aktiviert werden, ohne dass der Benutzer zur manuellen Aktivierung dieses Dienstes über eine spezielle Benutzerschnittstelle der Prozessmaschine aufgefordert werden muss.
- **Whitebox-Integration**: Bei dieser Integrationsform wird ein Werkzeug entweder speziell für den Einsatz in einer Umgebung neu entwickelt oder es steht bei existierenden Werkzeug der Quellcode zur Verfügung, um die erforderlichen Modifikationen zur Anbindung und Anpassung an die Leit- und Modellierungsdomäne vornehmen zu können. Konkret bedeutet dies, dass der Werkzeugentwickler von der Leit- und Modellierungsdomäne vorgegebene Schnittstellen, Protokolle, Schemata, Infrastrukturkomponenten u.ä. berücksichtigen kann, um eine nahtlose Integration zu erreichen.

A posteriori- und a priori-Integration

In der Literatur findet sich für die beiden erstgenannten Integrationsformen häufig auch der Begriff *a posteriori-Integration* (siehe z.B. [NaWe99]), da das betreffende Werkzeug nicht speziell für den Einsatz innerhalb einer prozesszentrierten Umgebung ausgelegt wurde, sondern erst *im nachhinein* (a posteriori) dort eingebunden wird. Entsprechend spricht man bei der Whitebox-Integration von *a priori-Integration*, da die Verwendung des Werkzeugs im Kontext einer prozesszentrierten Umgebung bereits bei seiner Entwicklung mit eingeplant wurde bzw. die Möglichkeit besteht, es in seiner internen Struktur anzupassen.

Lohnt sich die Beschäftigung mit a-priori-Integrationsansätzen überhaupt?

Es ist nicht zu verkennen, dass a priori-Integrationsansätze, vor allem solche, die im akademischen Umfeld entstehen und nicht durch die Marktposition großer Softwarefirmen (z.B. Microsoft) oder anerkannter Standardisierungsinitiativen (z.B. die OMG) getragen werden, *in der Praxis* häufig auf nur geringe Akzeptanz stoßen [BrEW92]. Dies mag zu einem wesentlichen Teil daran liegen, dass bei a priori entwickelten Werkzeugen lediglich ein spezifischer, aus wissenschaftlicher

Sicht interessanter Aspekt – hier: die Prozessintegration – im Vordergrund steht. Diese Fokussierung geht im Vergleich zu entsprechenden kommerziellen Werkzeugen fast immer auf Kosten des Funktionsumfangs und der Benutzbarkeit, da entsprechende Ressourcen zu deren Realisierung fehlen. Die wesentliche Kritik, die gegenüber a-priori-Integrationsansätzen ins Feld geführt wird, lautet also, dass der Aufwand für die Neuentwicklung *konkurrenzfähiger* Werkzeuge (bzw. für die Modifikation des Quellcodes existierender Werkzeuge, sofern überhaupt verfügbar) in keinem vernünftigen Verhältnis steht zum Vorteil, der sich aus der Integriertheit der Werkzeuge ergibt, und somit die Anwendung des Ansatzes prohibitiv teuer macht.

Mangelnde Akzeptanz akademischer Ansätze in der Praxis

Ein Beleg hierfür findet sich zum Beispiel in [EBLA96; EAMP97], wo die Autoren über ihre Erfahrungen aus einem Experiment zur Einführung von Prozesstechnologie bei der British Airways (BA) im Rahmen des GOODSTEP-Projekts [GOOD94] berichten. In dieser Studie wurden speziell die Prozesse zum Management von C++-Klassenbibliotheken bei BA untersucht und durch eine prozesszentrierte Umgebung unterstützt, welche mehrere, auf Basis des GTSL-Ansatzes ([Emme95; Emme96], siehe auch Abschnitt 3.3.4.2) neu entwickelte Werkzeuge umfasst. Als Resultat des Experiments konnten zwar wertvolle Erkenntnisse über die Prozessmodellierungssprache SLANG sowie über die Prozesse bei British Airways selbst gewonnen werden, die entstandene prozesszentrierte Umgebung inklusive der neuen Werkzeuge wurde jedoch nicht in den Produktivbetrieb bei British Airways übernommen. Als Hauptgrund wurde angeführt, dass der Austausch der bisherigen Umgebung durch die neue Umgebung eine zu radikale Umstellung für die Entwickler bedeutet hätte, die nicht auf die Benutzeroberfläche und den Funktionsumfang der ihnen vertrauten Werkzeuge verzichten wollten. Derartige Motivationen stehen hinter einer Reihe von a posteriori-Integrationsansätzen, wie sie bei den prozesszentrierten Umgebungen *Provence* [BaKr93; BaKr95], *Marvel/Oz* [Barg92; BeKa98] mit den Werkzeugintegrationsprotokollen *SEL* und *MTP* [VaKa96] oder auch den ingenieurwissenschaftlichen Integrationsprojekten *IMPROVE* [NaWe99] verfolgt werden.

Ungeachtet dieser Diskussion befassen wir uns in dieser Arbeit dennoch zunächst mit der a priori-Integration von Werkzeugen. Mit dem Ziel des wissenschaftlichen Erkenntnisgewinns geht es uns um die Charakterisierung der *Zielvorstellung* einer *idealtypischen* Prozessintegration von Werkzeugen, die über den bei heute existierenden Werkzeugen und Integrationsmethoden erreichbaren Integrationsgrad hinausreicht. Wir beschäftigen uns mit dieser Fragestellung jedoch nicht nur auf der konzeptionellen Ebene. Um die prinzipielle Umsetzbarkeit dieser Ziele zu demonstrieren, entwickeln wir einen Ansatz, der durch explizite Prozess- und Werkzeugmodelle und eine konkrete Architektur für prozessintegrierte Werkzeuge gekennzeichnet ist. Ein wesentlicher Beitrag besteht insbesondere in der Bereitstellung eines Implementationsrahmens für prozessintegrierte Werkzeuge, in dem wesentliche Aspekte des Integrationsansatzes als vorgefertigte Softwarekomponenten vorliegen und der somit die oben angesprochene Aufwandsschwelle für die Anwendung des Ansatzes beträchtlich absenkt.

Bei der Einordnung unseres Ansatzes in die oben skizzierten Integrationsszenarien muss zudem unterschieden werden zwischen der *initialen Erstellung* eines Werkzeugs und seiner Einbindung in *evolvierende Prozesse*. Wie in Abschnitt 3.1.4 deutlich gemacht wurde, ist Prozessintegration in erster Linie als *Adaptierbarkeit* eines Werkzeug an sich kontinuierlich verändernde Prozessdefinitionen zu verste-

hen. Bei einem reinen Whitebox-Ansatz resultiert jede Prozessänderung in einer Reprogrammierung der betroffenen Werkzeuge, um deren Prozessintegration wieder herzustellen. Dagegen sind Werkzeuge, so sie denn erst mal nach unserem Ansatz entwickelt worden sind, *per se* prozessintegrierbar und lassen sich im Sinne der Greybox-Integration über dann vorhandene Schnittstellen und formale Modelle an sich ändernde Prozesse anpassen. Dies entspricht der in Abschnitt 3.1.4 erhobenen Anforderung nach Modellierungsadaptabilität. Modifikationen an den Werkzeugen auf Programmcodebene sind dann nur noch erforderlich, wenn völlig neue domänenspezifische Grundfunktionalitäten benötigt werden. Dies wäre jedoch auch bei einem kommerziellen Werkzeug, das den in einem bestimmten Prozess benötigten Funktionsumfang nicht anbietet, nicht anders.

Nach der Klärung der idealtypischen Prozessintegration im Sinne eines a priori-Ansatzes stellt sich die darüber hinaus weisende Frage, ob nicht auch *existierende* Werkzeuge über Methoden der Greybox-Integration, also mithilfe entsprechender *Wrapper-Techniken*, prozessintegriert werden können. Dazu sei an dieser Stelle schon darauf hingewiesen, dass die bloße Existenz von programmierbaren Werkzeug-APIs noch nicht viel über die Prozessintegrierbarkeit des Werkzeugs aussagt; diese hängt vielmehr stark vom Funktionsumfang der programmierbaren Laufzeitschnittstellen ab. So spielen neben dem Aufruf elementarer Werkzeugdienste auch der Zugriff auf die Interaktionselemente (z.B. Menüs) und die Art der Datenpersistierung eine wichtige Rolle. Die Anforderungen an die erforderlichen Werkzeug-APIs werden wir in Abschnitt 7.4 genauer beleuchten.

Blackbox-Integration spielt in dieser Arbeit nur eine untergeordnete Rolle, da sich diese Integrationsmethode, wie oben bereits diskutiert, nur für „leichtgewichtige“ (*light-weight*), Batch-artige Werkzeuge eignet [EmFi96]. Die in dieser Arbeit betrachten kreativen Entwurfsprozesse sind jedoch eher durch den Einsatz „schwergewichtiger“ (heavy-weight) Werkzeuge mit einem hohen Interaktionsgrad gekennzeichnet.

3.3 Integrationsanforderungen in prozessintegrierten Umgebungen

3.3.1 Überblick

Für die Integration zwischen den Prozessdomänen haben wir insgesamt sechs Anforderungen identifiziert [PoWe97; Poh*99], welche die teilweise schon in [Wass90; ThNe92; FeOh91; Bro*94; ChNo92; DoFe94; Böhm98; BeMü99] genannten Integrationsfragestellungen in wesentlichen Punkten präzisieren und erweitern. Im Folgenden geben wir zunächst einen kurzen Überblick, ehe wir die einzelnen Anforderungen in den Abschnitten 3.3.2 bis 3.3.7 detailliert beleuchten und mögliche Lösungsansätze bewerten. Bei der Darstellung der Anforderungen orientieren wir uns an der bewährten Strukturierung in Daten-, Kontroll- und Präsentationsaspekte.

3.3.1.1 Datenintegration

Anforderung 1:
Datenintegration zwischen den Prozessdomänen

Die Notwendigkeit zur Datenintegration zwischen den Prozessdomänen resultiert aus der Tatsache, dass Werkzeuge nicht auf isolierten, disjunkten Datenbeständen arbeiten. Vielmehr legt das Prozessmodell eine Abfolge von Bearbeitungsschritten fest, die im Allgemeinen durch unterschiedliche Werkzeuge abgedeckt werden. Da normalerweise Daten von einem Bearbeitungsschritt zum nächsten weitergereicht werden, entstehen logische und physische Abhängigkeiten zwischen Werkzeugdaten. Dazu müssen Werkzeuge zunächst einmal prinzipiell in die Lage versetzt werden, untereinander (d.h. innerhalb der Durchführungsdomäne) und mit der Prozessmaschine (d.h. mit der Leitdomäne) Entwurfsergebnisse auszutauschen. Weiterhin muss bei der wechselseitigen Bearbeitung überlappender Datenbestände dafür gesorgt werden, dass die werkzeugübergreifende Integrität der Daten kontrolliert und gegebenenfalls gesichert wird. Mit der Datenintegration innerhalb einer prozessintegrierten Entwurfsumgebung verfolgt man also zwei primäre Ziele: *Dateninteroperabilität* und *Datenkonsistenz* (Abschnitt 3.3.2).

3.3.1.2 Kontrollintegration

Gegenstand der Kontrollintegration sind *Aufrufmechanismen* zur unmittelbaren Aktivierung und Steuerung von Werkzeugdiensten, ohne dass dazu direkte Eingriffe des Benutzers erforderlich sind, und *Notifikationsmechanismen*, mit denen ein Werkzeug andere Komponenten der Entwurfsumgebungen über Änderungen seines Zustands informieren kann [Wass90; ThNe92; BCTW96; Brow93]. Hierzu ist neben dem reinen Austausch von Entwurfsdaten, etwa über den Im- und Export von Dateien oder über ein gemeinsam genutztes Repository, eine *direkte Interaktion* zwischen den Werkzeugen untereinander sowie mit der Prozessmaschine zum Austausch von Kontroll- und Zustandsinformationen erforderlich.

Kontrollintegration = Prozessintegration ?

Die Dimension der Kontrollintegration betrachtet also vor allem dynamische Aspekte der Steuerbarkeit des Werkzeugverhaltens und steht damit in direktem Zusammenhang mit dem Kernziel der Prozessintegration. In der Tat werden in manchen Publikationen die Grenzen zwischen Kontroll- und Prozessintegration nicht ganz scharf gezogen (siehe z.B. [Kelt93; ScBr93]), wobei tendenziell Kontrollintegration als werkzeugübergreifende Programmiertechnik für die Automation kleinerer Arbeitseinheiten verstanden wird, während mit Prozessintegration die Abwicklung größerer Arbeitseinheiten in unterschiedlichen Werkzeugen verbunden wird. Hier besteht jedoch die Gefahr, dass unterschiedliche Integrationsbegriffe mit möglicherweise konkurrierenden Integrationsmechanismen für ähnliche Sachverhalte verwendet werden, was das Risiko inhomogener und schwer durchschaubarer Systemarchitekturen birgt. Andere Arbeiten wie etwa [Tull91; MiSc92] propagieren daher eine strikte Trennung zwischen Kontroll- und Prozessintegration. Sie weisen daraufhin, dass ein kontrollintegriertes Werkzeug, also eines, das über feingranular programmierbare Laufzeitschnittstellen durch andere Werkzeuge aufgerufen werden kann und interne Zustandsänderungen nach außen meldet, noch nicht unbedingt ein prozessintegriertes Werkzeug darstellt, solange keine *expliziten Prozessdefinitionen* das Werkzeugverhalten und die Interaktionen mit

anderen Werkzeugen festlegen. In dieser Arbeit schließen wir uns dieser Auffassung an: die Dimension der Kontrollintegration definiert somit lediglich Basisintegrationsdienste für die höhere Ebene der Prozessintegration. Das Zusammenschalten von Werkzeugen über Mechanismen der Kontrollintegration ist also erst dann als prozessintegriert zu betrachten, wenn die Interaktionsmuster zwischen den Werkzeugen auf expliziten und einfach änderbaren Prozessdefinitionen beruhen, die in der Modellierungsdomäne abgelegt sind.

Vor diesem Hintergrund sind bei der Auswahl und Ausgestaltung von Infrastrukturen, Mechanismen und Beschreibungstechniken für die Kontrollintegration drei Aspekte von besonderer Bedeutung:

Anforderung 2:
Prozessorientierte Mediation der Werkzeuginteraktionen

Hinter Werkzeuginteraktionen verbirgt sich inhärent Wissen über Arbeitsabläufe. Diese Interaktionsmuster sollten nicht in den beteiligten Werkzeugen hartkodiert sein, sondern für die Werkzeuge transparent in expliziten Prozessdefinitionen in der Modellierungsdomäne verankert sein. Für einen Kontrollintegrationsmechanismus bedeutet dies, dass eine Mediation über die Prozessmaschine einer direkten Werkzeuginteraktion vorzuziehen ist (Abschnitt 3.3.3)

Anforderung 3:
Konzeptuelle Beschreibung von Werkzeugdiensten

Um zur Modellierungszeit innerhalb einer Prozessdefinition auf Werkzeuge Bezug nehmen zu können, müssen deren Dienste in geeigneter Weise auf einer konzeptuellen Ebene, die von Implementierungsspezifika abstrahiert, repräsentiert werden (Abschnitt 3.3.4).

Anforderung 4:
Synchronisation der Prozessdomänen

Als wesentliche Voraussetzung für sinnvolle Prozessunterstützung muss der Zustand der Prozessmodellausführung in der Leitdomäne den tatsächlichen Prozesszustand in der Durchführungsdomäne möglichst exakt widerspiegeln. Hierzu sind oberhalb der durch einen Kontrollintegrationsmechanismus gegebenen Fähigkeit zur Interaktion geeignete *Synchronisationsprotokolle* zwischen den Domänen zu definieren (Abschnitt 3.3.5).

3.3.1.3 Präsentationsintegration

Präsentationsintegration befasst sich mit software-ergonomischen Fragestellungen bei der Verwendung unterschiedlicher Werkzeuge in der Arbeitsumgebung eines Entwicklers [Wand93]. Aus den Arbeitswissenschaften stammen Bewertungs- und Gestaltungskriterien wie Kompetenzförderlichkeit, Handlungsflexibilität und Aufgabenangemessenheit [VDI#90]. Diese implizieren eine Vereinheitlichung des Erscheinungsbilds der Werkzeuge (einheitliches „Look & Feel") und ein gemeinsames Interaktionsparadigma der zu integrierenden Werkzeuge. Gerade zum letztgenannten Punkt ergeben sich aus der Prozessorientierung zwei wesentliche Aspekte, die den Bereich der Präsentationsintegration berühren: die prozesssensi-

tive Anpassung der Interaktionsmöglichkeiten des Benutzer und der werkzeugunterstützte Aufruf von Prozessfragmenten.

Anforderung 5:
Prozesssensitive Anpassung der Interaktionsmöglichkeiten

Die Prozessdefinitionen und der aktuelle Prozesszustand definieren die zu einem bestimmten Zeitpunkt zu bearbeitenden Objekte und die zulässigen Aktionen auf den Objekten. Diese sollten sich in der Benutzeroberfläche der Werkzeugs widerspiegeln, d.h. ein Werkzeug sollte in der Lage sein, die Interaktionsmöglichkeiten des Benutzers prozesssensitiv anzupassen, indem der Zugriff auf Objekte und Kommandos an der Benutzeroberfläche dynamisch eingeschränkt wird (Abschnitt 3.3.6).

Anforderung 6:
Werkzeugunterstützter Aufruf von Prozessfragmenten

Die Werkzeug sollten den Entwickler beim Abgleich der im Werkzeug vorliegenden Prozesssituation mit den Prozessdefinitionen unterstützen und ggf. den Aufruf von Prozessfragmenten direkt aus der Benutzeroberfläche des Werkzeugs erlauben. Im Idealfall sollte für den Entwickler kein Unterschied zwischen der Aktivierung werkzeugeigener Dienste und extern definierter Prozessfragmente bestehen (Abschnitt 3.3.7).

3.3.2 Datenintegration zwischen den Prozessdomänen

3.3.2.1 Motivation

Werkzeuge erzeugen, konsumieren, modifizieren, transformieren und analysieren im Zuge des Entwurfsprozesses auf vielfältige Art und Weise Datenobjekte. Ein Teil der Daten ist nicht-persistent und nur für die Dauer eines Arbeitsschritts relevant. In der Regel reicht die Lebensdauer der Daten jedoch über die eines Bearbeitungsschrittes in einem einzelnen Werkzeug hinaus.

Die Notwendigkeit zur Datenintegration resultiert somit aus der Tatsache, dass Werkzeuge nicht auf isolierten, disjunkten Datenbeständen arbeiten. Vielmehr legt das Prozessmodell eine Abfolge von Bearbeitungsschritten fest, die im Allgemeinen durch unterschiedliche Werkzeuge abgedeckt werden, und induziert so logische und physische Abhängigkeiten zwischen Werkzeugdaten. Dazu müssen Werkzeuge zunächst einmal prinzipiell in die Lage versetzt werden, untereinander (d.h. innerhalb der Durchführungsdomäne) und mit der Prozessmaschine (d.h. mit der Leitdomäne) Entwurfsergebnisse in Form von Werkzeugdaten auszutauschen. Weiterhin muss bei der wechselseitigen Bearbeitung überlappender Datenbestände dafür gesorgt werden, dass die werkzeugübergreifende Integrität der Daten kontrolliert und gegebenenfalls gesichert wird. Mit der Datenintegration innerhalb einer prozessintegrierten Entwurfsumgebung verfolgt man also zwei primäre Ziele: *Dateninteroperabilität* und *Datenkonsistenz*.

Dateninteroperabilität

Unter Dateninteroperabilität verstehen wir die Fähigkeit, effizient und mit möglichst geringem Aufwand die Entwurfsergebnisse eines Werkzeugs anderen Werkzeugen sowie der Prozessmaschine in sinnvoller Weise zugänglich zu machen.

Stufen der Dateninteroperabilität

Die dabei auftretenden *Brüche* und entsprechend erforderlichen *Harmonisierungsmaßnahmen* kann man gemäß einem auf Brown und McDermid zurückgehenden Klassifikationsschema [BrMc91; BrEM92; Brow93] auf fünf unterschiedlichen Stufen charakterisieren:

- **Trägerstufe** (*carrier level*): Auf dieser untersten Ebene wird sichergestellt, dass ein Werkzeug bzw. die Prozessmaschine die Daten eines anderen Werkzeugs als uninterpretierten Zeichenstrom lesen kann. Integrationsmaßnahmen auf der Trägerstufe erfordern beispielsweise die Konversion einer Zeichencodierung (z.B. ASCII) in eine andere (z.B. UNICODE).
- **Lexikalische Stufe** (*lexical level*): Hier herrscht bei den zu integrierenden Werkzeugen ein gemeinsames Verständnis über die grundlegenden lexikalischen Einheiten. Ein Beispiel für Integration auf der lexikalischen Stufe stellen die frühen Unix-Werkzeuge `tbl`, `eqn`, und `pic` dar, die in der Documenter's Workbench [AT&T84] zusammengefasst sind. Bei diesen Werkzeugen ist zum Beispiel per Konvention festgelegt, dass Formatierungsanweisungen durch einen „.“ eingeleitet werden.
- **Syntaktische Stufe** (*syntactical level*): Auf der syntaktischen Stufe verfügen die zu integrierenden Werkzeuge über ein gemeinsames Verständnis der verwendeten Datenstrukturen bzw. der Regeln, nach denen die Datenstrukturen gebildet werden. Offensichtliche Beispiele für diese Integrationsstufe finden sich zum Beispiel in eng integrierten Programmierumgebungen (z.B. [ReTe81; HaNo86]), in denen die einzelnen Werkzeuge (syntaxgesteuerte Editoren, Compiler, Debugger) auf einheitlichen, internen Datenstrukturen (Symboltabelle, abstrakter Syntaxbaum) aufsetzen.
- **Semantische Stufe** (*semantical level*): Diese Ebene setzt voraus, dass die zu integrierenden Werkzeuge nicht nur gemeinsamen Konventionen bezüglich der verwendeten Datenstrukturen folgen, sondern auch die Bedeutung der Datenstrukturen in gleicher Weise interpretieren. Integration auf der semantischen Stufe kann auf zwei Arten erreicht werden. Zum einen können Datenstrukturen und Operationen auf diesen Datenstrukturen vorab spezifiziert werden, so dass Werkzeughersteller a priori über die Bedeutung der einzelnen Datenstrukturen, deren Bezeichnungen, die Auswirkungen von Operationen auf den Datenstrukturen usw. informiert sind. Zum anderen können Meta-Informationen über die Datenstrukturen und die zugehörigen Operationen in einem Repository verwaltet werden [Ber*99; JePa97; JePa97]. Der letztgenannte Ansatz hat den Vorteil der einfachen Erweiterbarkeit, da die Meta-Daten zur Laufzeit der Werkzeuge angefragt werden können.
- **Methodenstufe** (*method level*): Auf der obersten Stufe wird die Interaktion und damit der Datenaustausch und die Datenkonsistenz zwischen Werkzeugen im Kontext der zu unterstützenden Entwicklungsprozesse betrachtet. Die Kenntnis der Entwicklungsprozesse hilft, die auszutauschenden Daten anhand der werkzeugübergreifenden Abläufe zu identifizieren

und Politiken, unter welchen Umständen Datenkonsistenz durchgesetzt werden muss, festzulegen.

Im Kontext dieser Arbeit streben wir Datenintegration auf der Methodenstufe an, was insbesondere ein gemeinsames Verständnis zwischen Werkzeugen und Prozessmaschine über die Semantik der Daten impliziert, da der weitere Ablauf der Prozessmodellausführung sich häufig erst aus der Kenntnis und Interpretation der Daten ergibt, die von einem Werkzeug als Ergebnis eines Bearbeitungsschritts an die Prozessmaschine zurückgemeldet werden.

Datenkonsistenz

Ein zweites wesentliches Ziel der Datenintegration ist neben der reinen Dateninteroperabilität die Gewährleistung von Datenkonsistenz über Werkzeuggrenzen hinweg. Zwischen den Daten, auf denen unterschiedliche Werkzeuge arbeiten, existieren in einer integrierten Umgebung im Allgemeinen vielfältige Abhängigkeiten. Diese Abhängigkeiten können zum einen physischer Natur sein, wenn z.B. zwei Werkzeuge auf der gleichen Datenablage operieren. Logische Abhängigkeiten ergeben sich, wenn zwischen den Daten unterschiedlicher Werkzeuge inhaltliche Querbezüge existieren. Als einfaches Beispiel betrachten wir logische Datenabhängigkeiten innerhalb einer UML-Werkzeugumgebung: wenn in dem von einem Klassendiagramm-Editor verwalteten Klassenmodell der Name einer Klasse geändert wird, muss diese Änderung im korrespondierenden Zustandsdiagramm des Verhaltenseditors nachgezogen werden. Neben solchen formal modellierbaren, strukturellen Abhängigkeiten zwischen verschiedenen Produktdaten werden durch den Entwicklungsprozess auch „weiche“ Abhängigkeiten induziert (z.B. zwischen einer als Fließtext notierten Anforderungsspezifikation und einem darauf basierenden Klassenmodell), die bei der Sicherung der Datenkonsistenz gleichwohl zu berücksichtigen sind.

3.3.2.2 Bewertung existierender Ansätze

Bei der Betrachtung konkreter Datenintegrationsmechanismen lassen sich die Ziele Dateninteroperabilität und Datenkonsistenz nicht immer voneinander trennen, da eine gewählte Art des Datenaustausch mit bestimmten Techniken zur Konsistenzsicherung- und -kontrolle einher geht. Wir legen daher im Folgenden zunächst den Schwerpunkt auf Mechanismen zur Dateninteroperabilität zwischen Werkzeugen und diskutieren dann die Stärken und Schwächen in Hinblick auf die Sicherung und Kontrolle von Datenkonsistenz.

Für den Austausch von Daten zwischen Werkzeugen lassen sich im Wesentlichen drei Methoden unterscheiden ([ChNo92; Bro*94; BeDa94], vgl. Abb. 8): direkter Datenaustausch, Datei-basierter Datenaustausch und Repository-basierter Datenaustausch.

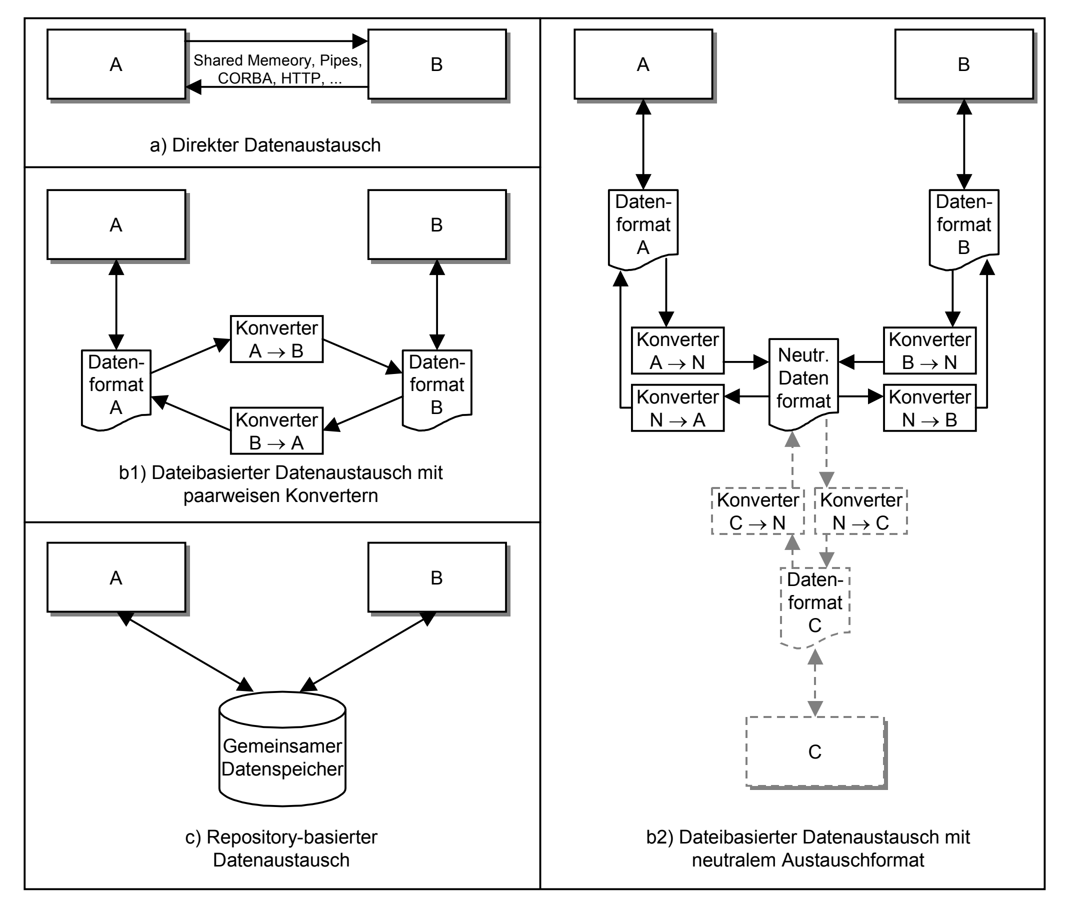


Abb. 8: *Datenaustausch-mechanismen*

Direkter Datenaustausch

Beim direkten Datenaustausch werden die (Ausgabe-)Daten eines Werkzeugs direkt an ein anderes geleitet, welches die Daten dann weiter verarbeitet (siehe Abb. 8a)[4]. Populär wurde diese Art der Datenintegration insbesondere in der *Unix Programmers Workbench*, wo sich durch das Zusammenschalten der Aus- und Eingabekanäle einfacher Batch-Werkzeuge (Filter, Compiler, Linker) komplexe und mächtige Transformationswerkzeuge bilden lassen [KeRi84]. Alternativ zu Shared Memory- oder Piping-Mechanismen können auch Kommunikationsmechanismen und -protokolle wie CORBA [OMG#97] oder HTTP [Fie*99] zum Datentransfer verwendet werden, so dass der Datenaustausch auch in einer räumlich verteilten, heterogenen Umgebung möglich ist. Die zeitlich getrennte Bearbeitung ist jedoch beim direkten Datenaustausch ebenso wie die Integration von mehr als zwei Werkzeugen mit Schwierigkeiten verbunden. Außerdem lässt sich die werkzeugübergreifende Konsistenzsicherung überlappender Daten nur schwer realisieren, u.a. deswegen, weil Daten im Allgemeinen nur als uninterpretierte Zeichenströme, d.h. auf der Trägerebene oder bestenfalls der lexikalischen Ebene (s.o.), ausgetauscht werden und es keine globale Sicht auf die Daten gibt.

[4] Wir diskutieren zunächst nur den Werkzeug-zu-Werkzeug-Datenaustausch. In der Abb. 7 bezeichnen A, B und C aber beliebige datenhaltende Komponenten, also Werkzeuge oder Prozessmaschine.

Datei-basierter Datenaustausch:

In der Praxis ist der Datei-basierte Austausch von Entwurfsdaten immer noch die am weitesten verbreitete Lösung. Werkzeuge legen ihre Daten in Dateien (eines gemeinsam benutzten Dateisystems) ab, die dann von anderen Werkzeugen gelesen werden. Dies erfordert, dass sich die Hersteller der Werkzeuge auf ein gemeinsames Datenformat verständigt haben. Da dies bei unabhängig voneinander entwickelten Werkzeugen jedoch im Allgemeinen nicht der Fall ist, werden zusätzliche Konvertierungsprogramme benötigt, die die von einem Werkzeug exportierten Daten in das Format eines importierenden Werkzeugs transformieren. Diese Vorgehensweise ist jedoch sehr mühsam und aufwändig, da zum einen die von den Werkzeugen verwendeten Datenformate und -schemata häufig nicht allgemein bekannt sind, so dass in diesen Fällen keine oder nur eine teilweise Transformation der Daten gelingt. Zum anderen werden für die paarweise, beidseitige Integration von n Werkzeugen insgesamt $2 * (n * (n - 1) / 2) \approx n^2$ Konvertierungsprogramme benötigt (siehe Abb. 8b1).

Das n^2-Konverter-Problem

Aus diesem Grunde sind in unterschiedlichen Entwurfsdomänen zahlreiche Standardisierungsinitiativen unternommen worden mit dem Ziel, geeignete Austauschformate zu entwickeln und zu vereinheitlichen. Beim Vorliegen eines von allen Werkzeugherstellern einer Entwurfsdomäne respektierten Austauschstandards wird für jedes Werkzeug lediglich ein Import- und ein Exportfilter benötigt, um Daten aus dem neutralen Datenformat in das proprietäre Werkzeugformat zu konvertieren und umgekehrt (siehe Abb. 8b2). Somit reduziert sich die Anzahl der benötigten Konvertierungsprogramme auf $2 * n$.

Standardisierte Informationsmodelle

CDIF: CASE Tool Data Interchange Format

Im Bereich der Softwaretechnik liegt mit dem von der Electronics Industry Association getragenen CASE Data Interchange Format (CDIF) [Park92; EIA#94; Tann94] ein Standardisierungsversuch mit recht langer Tradition vor. Ähnlich dem weiter unten beschriebenen IRDS-Rahmenwerk basiert CDIF auf einer mehrstufigen Modellhierarchie, die eine flexible Erweiterung des Standards zum Ziel hat. Auf der obersten Ebene (Metametamodell) werden die prinzipiellen Strukturen und Konstrukte für die Spezifikation von Modellierungstechniken festgelegt. Mithilfe dieser Konstrukte werden auf der nächsten Ebene Schemata (Metamodelle) für bestimmte Modellierungstechniken (z.B. Datenflussdiagramme) spezifiziert. Auf der Modellebene sind die eigentlichen Werkzeugdaten als Instanzen der Metamodelle angesiedelt. Bei einem Datentransfer werden sowohl die Modelldaten (also z.B. ein konkretes Datenflussdiagramm) als auch die zugehörigen Metamodelldaten zwischen den beteiligten Werkzeugen gemäß einer vom CDIF-Standard festgelegten Klartext-Syntax ausgetauscht. Durch die Übermittlung des Metamodells erhält das Empfängerwerkzeug Informationen für die sinnvolle Interpretation der Modelldaten, was insbesondere für Metamodell-Erweiterungen, die über den CDIF-Standard hinausgehen, wichtig ist.

Insgesamt hat CDIF als spezifisches Austauschformat von Seiten der Werkzeug-Hersteller nur zögerliche Unterstützung erfahren und spielt als konkreter Austauschstandard heute keine Rolle mehr. Allerdings finden sich wesentliche Elemente des CDIF-Metametamodells in der Meta Object Facility (MOF) der

Meta Object Facility der OMG

Object Management Group[5] (OMG) wieder [OMG#97c]. MOF ist ein von der OMG standardisierter Mechanismus zur Darstellung von Metadaten und wurde insbesondere für die Metamodellierung der ebenfalls zum OMG-Standard erhobenen Unified Modeling Language verwendet. MOF wird verkörpert durch einen Satz von CORBA-APIs, über die Werkzeuge den Austausch von UML-Daten realisieren können. In eine ähnliche Richtung geht auch das Open Information Model im Microsoft-Repository [Ber*99], das als eine Sammlung von definierten COM-Schnittstellen für den Austausch von Modell- und Metamodelldaten organisiert ist.

Als Alternative zu API-basierten Methoden des Datenaustauschs gewinnt in jüngster Zeit die HTML-Erweiterung XML (Extended Markup Language [BrPS98]) immer größere Bedeutung sowohl als Beschreibungssprache für einheitliche Austauschformate als auch zur Darstellung konkreter Austauschdaten. Mit *XMI*[6] [XMI#99] liegt mittlerweile ein OMG-Standard vor, der die Abbildung von MOF bzw. MOF-kompatiblen UML-Modellen in eine XML-basierte Darstellung beschreibt. XMI wurde in relativ kurzer Zeit auf breiter Front von wichtigen Werkzeug-Herstellern aufgegriffen (z.B. Rational Rose, Together und Argo UML) und scheint sich als *der* Austauschstandard für UML-Modelle durchzusetzen. Bemerkenswert an dieser Initiative ist, dass mit XMI im Gegensatz zur bisherigen OMG-Philosophie ein komplementärer Mechanismus zum direkten, CORBA-basierten Datenaustausch in die Standardisierungsbemühungen der OMG aufgenommen wurde. In [DHTT00] wird der XMI-basierte Datenaustausch an einem Fallbeispiel untersucht und mit dem direkten Datenaustausch über Werkzeug-APIs verglichen. Hier kommen die Autoren zum Schluss, dass sich die XMI-basierte Methode primär in *asynchronen* Kooperationssituationen eignet, während die kontinuierliche, inkrementelle Synchronisation zwischen verschiedenen UML-Werkzeugen am besten über den direkten Datenaustausch und Ereignisnotifikationsmechanismen mithilfe entsprechender Werkzeug-APIs (siehe auch Abschnitt 3.3.3) zu realisieren ist.

XMI: ein XML-basierter Austauschstandard für UML-Modelle

Die Konsistenzsicherung in Datenintegrationsansätzen, die auf Dateiaustausch basieren, stellt ein massives Problem dar, da die einzelnen Werkzeuge lediglich eine lokale Sicht auf die Daten haben und keine werkzeugübergreifende Kontrolle der Gesamtkonfiguration der Daten auf der feingranularen Ebene existiert [Kipe94; Meye91]. Ein konkreter Bezug zwischen den Daten unterschiedlicher Werkzeug wird immer nur zum Zeitpunkt der Transformation der Daten von einem Werkzeug zu einem anderen (mithilfe eines Konverters) hergestellt. Nachfolgende Bearbeitungen der Daten in einem der betroffenen Werkzeuge können Inkonsistenzen hervorrufen und sind bei einer manuellen Vorgehensweise nicht ohne weiteres zu entdecken und zu beheben.

Konsistenzsicherung beim Dateiaustausch

Zur Unterstützung der Konsistenzsicherung voneinander abhängiger Daten in unterschiedlichen Werkzeugen sind im Rahmen des IPSEN-Projekts [Nagl96] eine Reihe von *Integratoren* [Lefe95] entwickelt worden, die im Kern als Erweiterung traditioneller Datei-Konverter verstanden werden können. Der wesentliche Unter-

Integratoren des IPSEN-Projekts

---

[5] Die Object Management Group ist ein Herstellerkonsortium mit über 700 Mitgliedern (darunter praktisch jedes namhafte Unternehmen der IT-Branche mit Ausnahme von Microsoft), das sich der Förderung und Etablierung objektorientierter Techniken in verteilten, heterogenen Systemumgebungen zum Ziel gesetzt hat.

[6] XML Metadata Interchange

schied zu einfachen Konvertern besteht in der fortlaufenden Wartung feingranularer Beziehungen zwischen Inkrementen unterschiedlicher Dokumente[7] über eine initiale Transformation hinaus. Dazu werden Korrespondenzbeziehungen zwischen Werkzeugdokumenten innerhalb eines eigenen Integrationsdokuments verwaltet, anhand dessen Änderungen in den beteiligten Werkzeugdokumenten automatisch oder mit Benutzerinteraktion nachgezogen werden können. Die in [Lefe95] vorgestellten Integratoren sind auf die Konsistenthaltung von jeweils zwei Dokumenten (und somit maximal zwei Dokumenttypen) beschränkt. Mit zunehmender Anzahl unterschiedlicher Dokumenttypen steigt somit auch der Bedarf an Integratoren zur Konsistenthaltung der Dokumente untereinander. Dies ist mit einem erheblichen Realisierungsaufwand verbunden, auch wenn sich generische Teile eines Integrators in einem Rahmenwerk verallgemeinern lassen. Nicht unproblematisch ist auch die Koordination mehrerer Integratoren bei einem komplexen Geflecht unterschiedlicher Dokumenttypen. Die Konsistenthaltung von Daten ist selbst ein Prozess, der projekt- und organisationsspezifisch entsprechend unterschiedlicher Strategien und Konsistenzbegriffe ausgestaltet werden sollte. Diese Prozesse sind in den Integratoren jedoch als komplexe Kommandos kodiert und können nicht ohne weiteres abgeändert und auf das Zusammenspiel mehrerer Integratoren abgestimmt werden.

Repository-basierter Datenaustausch

Bei dieser klassischen, schon im ursprünglichen Stoneman-Report [DoD#80] propagierten Datenintegrationsmethode tauschen die Werkzeuge untereinander und mit der Prozessmaschine gemeinsame Entwurfsdaten über eine (logisch) zentrale Datenbank aus (siehe Abb. 8c), die im Kontext von Entwurfsumgebungen meist als *Repository* bezeichnet wird [BeDa94; Bro*94; Sche93; Tann94; Ortn99; IRDS90]. Der Verwaltung und der Austausch von Daten über ein Repository weist im Vergleich zum Datenaustausch über Dateien – selbst beim Vorliegen kanonischer, standardisierter Datenformate wie CDIF oder XMI oder bei der Nutzung von Integratoren – eine Reihe gravierender Vorteile auf [BeDa94]:

- Der Aufbewahrungsort der Daten ist bekannt: sie befinden sich innerhalb der Repositories und sind nicht über mehrere Dateien verteilt, welche von den einzelnen Werkzeugen unabhängig verwaltet werden müssen;
- Es gibt lediglich eine Kopie von jedem geteilten Objekt, so dass Inkonsistenzen durch die unabhängige Manipulation und Verwaltung mehrerer Kopien eines Objekts in unterschiedlichen Werkzeugen gar nicht auftreten können;
- Information geht beim Datenaustausch zwischen Werkzeugen nicht verloren, denn selbst wenn ein Werkzeug nicht alle spezifischen Daten eines anderen versteht, bleiben diese Informationen im Repository erhalten;
- Die Kontrolle der Datenobjekte erfolgt werkzeugübergreifend auf einheitliche Weise (Versionierungsmodell, Konfigurationsmodell);

[7] In der IPSEN-Terminologie wird stets von einem *Dokumentbegriff* ausgegangen, der eine aus *administrativer* Sicht mehr oder weniger natürliche Zusammenfassung zusammengehöriger Daten darstellt, als abstrakter Datentyp modelliert wird und häufig mit dem Begriff einer physischen Datei zusammenfällt.

- Geteilte Datenobjekte können inkrementell aktualisiert werden im Gegensatz zum direkten bzw. Datei-basierten Datenaustausch, der inhärent ein Batch-artiger Prozess ist;
- Daten können werkzeugübergreifend und selektiv angefragt werden, um z.B. die Revisionshistorie eines Objekts oder die Menge der von einem Objekt abhängigen Objekte zu finden.

Werkzeuge, die ihre Daten innerhalb des Repositories verwalten, profitieren also bereits von traditionellen Datenbankfunktionalitäten: einem *einheitlichen Datenmodell* (zur Strukturierung der Daten), *Anfragen* (zum selektiven Zugriff auf den Repository-Inhalt), *Sichten* (um die Datenunabhängigkeit zwischen den auf dem Repository arbeitenden Werkzeugen zu erhöhen), *Integritätskontrolle* (zur Konsistenzsicherung), *Zugriffskontrolle* und *Transaktionen*. Darüber hinaus werden von einem Repository aber noch zusätzliche Funktionalitäten gefordert, die über reine Datenbankfunktionalität hinausgehen. Dazu gehören unter anderem die Verwaltung *lokaler Arbeitskontexte* (zur Abschirmung einzelner Entwickler) und *check-in/check-out-Operationen* (zum Abgleich lokaler Arbeitskontexte mit dem globalen Entwurfsdatenbestand), das *Versions- und Konfigurationsmanagement* sowie *Notifikationsdienste* für die Propagation von Änderungen an interessierte Klienten.

Abstraktionsebenen innerhalb eines Repositories

Ein wichtiges Charakteristikum von Repositories ist weiterhin, dass sich die von den Werkzeugen gemeinsam verwendeten Informationen nicht nur auf die eigentlichen Daten beschränken, sondern auch Metadaten (verwendete Datenschemata, z.B. spezielle Datensatz- und -felddefinitionen), Metametadaten (verwendete Datenmodelle, d.h. erlaubte Formate für Datensatz- und Felddefinitionen) usw. umfassen. Durch die integrierte Verwaltung von Daten unterschiedlicher Ebenen unterscheiden sich Repositories von Data-Dictionary-Systemen, die typischerweise nur Metadaten, d.h. Datenbankschemata, verwalten. Prinzipiell impliziert der Begriff eines Repositories keine Beschränkung der Anzahl der Metaebenen. Allerdings konnten Kotteman und Konsynski [KoKo84] zeigen, dass vier Ebenen ausreichen, um sowohl die *Verwendung* und als auch *Evolution* eines Entwurfs-Informationssystems in integrierter Weise adäquat erfassen zu können. Eine ähnliche Beobachtung liegt auch der Architektur des ISO Information Resource Dictionary Standard IRDS [IRDS90; Byrn96] zugrunde, die in Abb. 9 dargestellt ist.

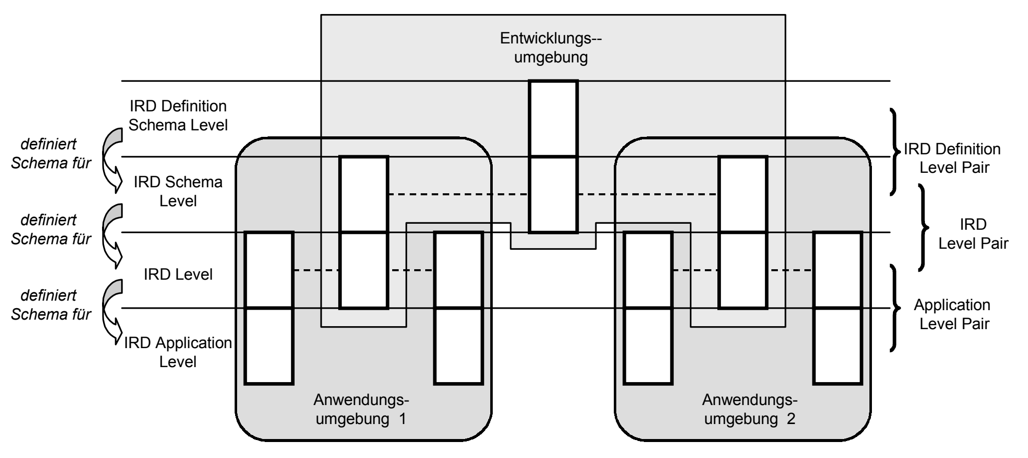


***Abb. 9**: Integration von Entwicklungs- und Anwendungs-umgebungen im IRDS-Rahmenwerk*

Ziel der IRDS-Architektur ist die Verknüpfung von Daten über eine (verteilte) *Benutzung* von Anwendungssystemen mit den Daten über die (verteilte) *Entwicklung*

von Applikationen. Dazu werden die Daten, die bei der Anwendungsbenutzung und -entwicklung anfallen, innerhalb von vier Instanziierungsebenen organisiert, die wir im Folgenden von unten nach oben skizzieren:

- Der **IRD Application Level** dokumentiert die Nutzung eines Anwendungssystems, d.h. er umfasst alle konkreten Produkt- und Ausführungsdaten, die bei der Verwendung eines Anwendungssystems entstehen. Diese Ebene korrespondiert mit der Instanzenebenen Klassen-basierter (Programmier-)Sprachen.
- Der **IRD Level** dokumentiert die Entwurfs- und Entwicklungsergebnisse bei der Erstellung eines Anwendungssystems, d.h. er beinhaltet Datenbank-Schemata und Anwendungsprogramme sowie alle Zwischenprodukte (Anforderungsspezifikationen, Architekturdiagramme etc.) und Beschreibung manueller Aktivitäten (Workflow-Beschreibungen). Auf dem IRD Level sind auch Prozessspuren, d.h. Daten, die die bei der Anwendungserstellung ausgeführten Entwicklungstätigkeiten dokumentieren, angesiedelt. Der IRD Level entspricht der Klassenebene Klassen-basierter Sprachen.
- Der **IRD Definition Level** spezifiziert die Sprachen, in der Schemata, Anwendungsprogramme und intermediäre Spezifikationen definiert werden. Auf dieser Ebene werden außerdem mögliche statische und dynamische Beziehungen zwischen den Sprachen definiert, etwa in Form von Sprachabbildungen (z.B. Balancierungs- und Transformationsregeln zwischen Sprachen für Anforderungs- und Entwurfsspezifikationen [Your89; Jann92; Kron92]) oder Prozessmodellen für die integrierte, konsistente Verwendung der Sprachen (z.B. Arbeitsabläufe für das Nachziehen einer Änderung eines ER-Schema in korrespondieren DFD-Diagrammen [Poh*99]). Der IRD Definition Level korrespondiert mit der Metaklassenebene.
- Der **IRD Definition Schema Level** definiert ein Metameta-Modell, mit dessen Hilfe die Objekte des IRD Definition Level beschrieben und miteinander in Beziehung gesetzt werden können.

Wie in Abb. 9 angedeutet, sind die vier Ebenen in miteinander verzahnten Ebenenpaaren angeordnet. Intuitiv kann ein Ebenenpaar als eine Datenbank verstanden werden, wobei die obere Ebene dem Schema und die untere Ebene dem Datenbank-Zustand entspricht. Innerhalb der IRDS-Architektur werden die Ebenenpaare so miteinander verschränkt, dass die Schemata der unteren Ebenenpaare durch den Datenbankzustand eines Ebenenpaars auf der nächst höheren Ebene koordiniert werden. Auf diese Weise entsteht eine verteilte Datenbank, in der:

- **Application Level Pairs** traditionellen Anwendungsdatenbanken entsprechen, die aus einem Anwendungsschema und einem Datenbankzustand bestehen;
- **IRD Level Pairs** mit Data Dictionaries oder Meta-Datenbanken korrespondieren. Zur Laufzeit fungieren IRD Level Paare als Koordinationsstellen innerhalb eines verteilten Informationssystems, zur Entwurfszeit dienen sie als die eigentlichen Entwurfsdatenbanken, über die die Entwurfswerkzeuge einer Entwicklungsumgebung integriert werden;

- **IRD Definition Level Pairs** dem gleichen Zweck wie IRD Level Pairs dienen, allerdings in Bezug auf die Evolution der Entwicklungsumgebungen selbst.

Während die miteinander verzahnten Application Level und IRD Level Pairs eine verteilte *Anwendungsumgebung* bilden, stellen die miteinander verzahnten IRD Level und IRD Definition Level Paare eine verteilte *Entwicklungsumgebung* dar. Auf diese Weise stellt die IRDS Architektur die prinzipiellen Konzepte bereit, um sowohl Informationen über die *Nutzung* als auch die *Entwicklung* eines verteilten Informationssystems miteinander zu integrieren.

Datengranularität

Bei dem Repository-Ansatz sind wir bisher davon ausgegangen, dass Werkzeuge und Prozessmaschine jeweils spezifische Sichten auf ein gemeinsames, globales Repository-Schema haben. Dies erlaubt eine größtmögliche Integration der Werkzeugdaten und der für die Leitdomäne relevanten Daten auch auf der feingranularen Ebene (siehe Abb. 10a).

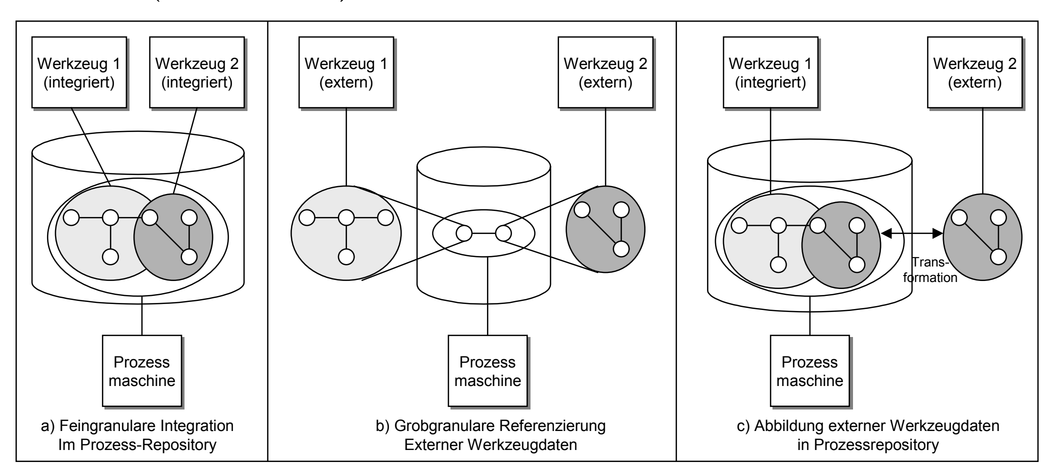


***Abb. 10**: Integration von Werkzeugdaten im Prozess-Repository*

Mit Ausnahme einiger weniger datenbankzentrierter Ansätze (z.B. GTSL [Emme95; Emme96]) wird dieses Idealbild in den meisten prozesszentrierten Umgebungen und Workflow-Managementsystemen nicht erreicht. Hier steht vielmehr die a-posteriori-Integration von Fremdwerkzeugen im Vordergrund, die ihre Daten in eigenen, vom Prozess-Repository unabhängigen Datenspeichern (typischerweise Dateien) ablegen. Die übliche Lösung, die z.B. in SPADE [BBFL94; BaDF96], Dynamite [HJKW96; HeKW97], Marvel [BaKa91; Barg92], Provence [BaKr93] u.a. praktiziert wird, besteht darin, dass die Werkzeugdaten aus Sicht des Prozess-Repositories nur grobgranular auf Dokumentebene modelliert und referenziert werden (siehe Abb. 10b). Der Inhalt dieser Dokumente ist nur den jeweiligen Werkzeugen zugänglich und kann nicht von der Prozessmaschine bei der Prozessmodellinterpretation berücksichtigt werden. Für eine administrative Sichtweise auf den Entwicklungsprozess reicht dies in der Regel aus. Die für die feingranulare Entwicklerunterstützung relevanten Daten befinden sich jedoch in der Regel unterhalb der Dokumentebene. Für die Prozessunterstützung zur Spezifikation von ER-Diagrammen ist es beispielsweise wichtig, ob sich im aktuellen ER-Diagramme noch unverbundene Entitätstypen befinden.

Eine Alternative zu beiden vorgenannten Integrationsarten besteht darin, dass die Daten von Fremdwerkzeugen über Transformatoren oder die oben bespro-

chenen Integrationswerkzeuge auf feingranularer Ebene in das Prozess-Repository abgebildet werden. Zusammen mit Werkzeugen, die ihre Daten direkt im Repository ablegen, ergäbe sich dann die in Abb. Abb. 10c dargestellte Situation.

Datenmodellierung

Die Daten in einem Repository können nach unterschiedlichen Datenmodellen strukturiert sein. Es lassen sich unter anderem folgende Kategorien von Datenmodellen unterscheiden:

- **Relational**: im relationalen Datenmodell werden Daten als Tupel in Tabellen strukturiert.
- **Graphbasiert**: graphbasierte Ansätze bilden Objekte als attributierte Knoten und Beziehungen zwischen Objekten als Kanten eines Graph ab. Ein häufiges Einsatzgebiet graphbasierter Datenmodelle und Datenbanken [KiSW95] ist die Abbildung abstrakter Syntaxbäume [Eng*92; And*99].
- **Objektorientiert**: Objektorientierte Datenmodelle [Cat*00; KeMo93] erweitern Modellierungskonzepte wie Objektidentität, Kasperung von Struktur und Verhalten, Vererbung und Polymorphie um den Aspekt der Persistenz. Strukturell objektorientierte Datenmodelle wie z.B. PCTE [WaJo93] und STEP/Express [ISO#94] aus dem ingenieurwissenschaftlichen Umfeld erlauben keine Methoden und haben ihren Ursprung in um komplexe Strukturen und Vererbung erweiterten Entity-Relationship-Modellen [Chen76; ElNa99]. In dem objektorientiert-logikbasierten Datenmodell O-Telos [Jeus92] übernehmen deklarativ formulierte Regel und Integritätsbedingungen die Rolle von Methoden. O-Telos eignet sich aufgrund seiner beliebig erweiterbaren Klassikationshierarchie besonders zur Metamodellierung.

Generell kann festgestellt werden, dass das relationale Datenmodell gravierende Defizite aufweist hinsichtlich einer adäquaten Darstellung komplexer Strukturen, wie sie in Entwurfsanwendungen häufig auftreten. Andererseits haben Datenbankmanagementsysteme für nicht-relationale Datenmodelle den Durchbruch bislang nicht geschafft, nicht zuletzt aufgrund immer noch mangelnder Performanz und Skalierbarkeit. Einen Kompromiss zwischen Modellierungsangemessenheit und Leistung stellen objekt-relationale Systeme dar. In diesen Systemen liegen Daten zwar immer noch in Tabellenform vor, es können jedoch zusätzliche benutzerdefinierte, komplexe Datentypen eingeführt werden.

Transaktionskontrolle

Ein Aspekt der Datenhaltung, der durch die Prozessintegration einer Entwurfsumgebung unmittelbar berührt wird, ist die Transaktionskontrolle. Ziel ist hier die Behandlung eines Prozessschritts oder einer Folge von Schritten als eine atomare Ausführungseinheit. Entwurfsschritte mit einer Vielzahl von Datenoperationen sollen gekapselt, von den Auswirkungen paralleler Schritte abgeschirmt und im Fehlerfall als Ganzes oder in kontrollierbaren Teilabschnitten zurückgesetzt werden. Im Gegensatz zu Transaktionen in Standard-Anwendungen (Bank-Applikationen, Buchungssystem u.ä.) sind Entwurfsprozesse durch langlebige, komplexe und interaktive Aufgaben charakterisiert, für die sich die ACID-Eigenschaften und Serialisierbarkeitsbegriffe aus Standarddatenbank-Anwendungen als ungeeignet

erwiesen haben. Erweiterte Transaktionsmodelle, die die klassischen Serialisierbarkeitsbegriffe aufweichen und dazu das in einer prozessintegrierten Umgebung verfügbare zusätzliche Prozesswissen ausnutzen, sind Gegenstand des umfangreichen Gebiets der *transactional workflows*. Aus Aufwandsgründen können wir jedoch diesen Aspekt in der vorliegenden Arbeit nicht vertiefen und verweisen stattdessen auf weiterführende Literatur [Bre*93; GeHo94; Alo*96; PuTV97].

3.3.2.3 Fazit

Datenintegration beschäftigt sich mit Fragen des Datenaustauschs und der Konsistenzsicherung von Daten in einer Entwicklungsumgebung. Es ist das am intensivsten untersuchte Integrationsproblem, für das mittlerweile ausgereifte Lösungsansätze vorliegen.

Am weitesten verbreitet ist der Austausch von Daten über Dateien. Da die Daten in unterschiedlichen Werkzeugen in der Regel in unterschiedlichen Formaten vorliegen, ist ein beträchtlicher Aufwand für Dateikonverter erforderlich, woran auch standardisierte Austauschformate wenig ändern. Da die Leitdomäne mit potenziell allen Werkzeugen interagieren muss, stellt eine auf Dateiaustausch basierende Integrationsmethodik eine wenig Erfolg versprechende Lösung dar. Zudem ist die Konsistenzsicherung bei diesem Verfahren sehr schwierig.

Die ideale Lösung zur Datenintegration in einer prozessintegrierten Umgebung besteht daher in einem (logisch) zentralen Repository, in dem die Daten der Werkzeuge und der Prozessmaschine unter einem globalen Schema integriert sind. Wegen der Speicherung der Daten an einem zentralen Ort entfallen als Konsistenzprobleme, die sich aus eine Replikation von Daten in unterschiedlichen Dateien ergeben könnten. Wegen der Notwendigkeit eines gemeinsamen Schemas funktioniert dieser Integrationsansatz allerdings nur bei der a priori-Integration von Werkzeugen.

Für die a posteriori-Integration unabhängig voneinander entstandener Werkzeuge ist eine unmittelbare Integration über ein Prozess-Repository in der Regel nicht möglich. Dies würde eine Einigung auf ein gemeinsames Datenmodell (für die syntaktische Integration) und ein globales Schema (für die semantische Integration) voraussetzen. Das Scheitern diverser Standardisierungsinitiativen in den 80er und 90er Jahren hat gezeigt, dass allgemeine Standards von den Werkzeugherstellern häufig als zu komplex und schwerfällig betrachtet werden und daher ignoriert werden. In solchen Fällen müssen die Werkzeugen aus den proprietären Datenspeichern entweder in das Repository abgebildet werden oder können nur grobgranular referenziert werden (etwa auf Dokumentebene).

3.3.3 Prozessorientierte Mediation der Werkzeuginteraktionen

3.3.3.1 Motivation

Ein grundlegendes Ziel von Mechanismen zur Kontrollintegration besteht in der programmgesteuerten Wiederverwendung und Rekombination elementarer Werkzeugfunktionalitäten in immer neuen Anwendungszusammenhängen durch gegenseitige Dienstbereitstellung und -nutzung. Entscheidend ist vor dem Hintergrund

einer prozessorientierten Anpassbarkeit die Beobachtung, dass durch jede Verschaltung atomarer Werkzeugdienste zu komplexeren Diensten letztendlich Abläufe spezifiziert werden, in denen potenziell prozessrelevantes Wissen hinterlegt ist und die somit direkt auf die Arbeitsmethodik des Benutzers rückwirken. Hierbei ist es zunächst unerheblich, ob die miteinander kombinierten Dienste von ein und dem selben Werkzeug stammen oder über Werkzeuggrenzen hinweg miteinander verkettet werden.

Prozessrelevante Werkzeugbeziehungen sollten offengelegt werden

Die Komposition von elementaren zu komplexen Diensten erfolgt durch den *koordinierten Aufruf* von Werkzeugdiensten, der über geeignete Kommunikationsmechanismen zu realisieren ist. In Hinblick auf die Adaptabilität an sich ändernde Prozesse ist bei der Bewertung existierender Lösungsansätze die Frage, wie fest Aufrufbeziehungen zwischen den Werkzeugdiensten in den Werkzeugen selbst „zementiert" werden, von entscheidender Bedeutung. Je transparenter für ein Werkzeug die Kopplung seiner Dienste mit denen anderer Werkzeuge ist, desto flexibler lässt sich ein Werkzeug in unterschiedlichen Prozessen einsetzen.

Bei der Realisierung (verteilter) Werkzeuginteraktionen lassen sich drei grundlegende Entwurfsparadigmen oder Architekturstile [Sha*95; ShGa96; AbAG95; LiKS99] differenzieren (siehe Abb. 11):

- **Expliziter Aufruf (Client-Server)**: Beim traditionellen Client-Server-Prinzip wird die Interaktion explizit durch ein Werkzeug initiiert, das sich als Klient der Dienste eines Server-Werkzeugs bedient. In einer heterogenen, verteilten Umgebung wird dieser Ansatz durch Middleware-Mechanismen wie *RPC*, *Object Request Broker* sowie *Namens- und Tradingdienste* verkörpert, die im Wesentlichen für eine Verteilungs- und Ortstransparenz sowie die Unabhängigkeit von den zugrunde liegenden Programmiersprachen sorgen. Die Interaktionsbeziehungen zwischen Werkzeugen haben den Charakter expliziter *Kommandoaufrufe*. Das prozessrelevante Ablaufwissen manifestiert sich in den Codefragmenten der jeweils aufrufenden Komponenten.
- **Impliziter Aufruf (ereignisbasiert)**: In ereignisbasierten Architekturen ruft ein Werkzeug nicht direkt die Dienste anderer Werkzeuge auf, sondern gibt lediglich Änderungen seines Zustands als *Notifikationen* über einen zentralen *Message-Server* an die Außenwelt bekannt und löst damit *implizit* entsprechende Reaktionen bei den für ihn anonymen Empfängern aus. Die Grundidee besteht also darin, dass bei Werkzeuginteraktionen nicht der Produzent, sondern der Empfänger einer Nachricht die Verantwortung für den Aufruf der „richtigen" Dienste trägt, so dass der Kontrollfluss quasi umgedreht wird.
- **(Prozess-)Mediierter Aufruf**: Der Nachteil der bisher skizzierten Ansätze besteht darin, dass prozessrelevantes Wissen über die Reaktion auf bestimmte Ereignisse in den Werkzeugen selbst verankert ist (im Sender beim expliziten Aufruf bzw. im Empfänger beim impliziten Aufruf). *Mediatorbasierte* Ansätze streben eine Trennung von Werkzeugen einerseits und Interaktionsbeziehungen zwischen Werkzeugen andererseits an. Werkzeuge exportieren Notifikationen über eigene Zustandsänderungen und nehmen Kommandoaufrufe von außen entgegen, reagieren aber nicht selbst auf Notifikationen und rufen keine Kommandos anderer Werkzeuge auf. Die Umsetzung von Notifikationen auf Kommandoaufrufe ist in einer

eigenen Komponente, dem *Mediator*, gekapselt. Der Mediatoransatz ist ein allgemeines Prinzip zur losen, d.h. erweiterbaren und adaptablen Kopplung von Softwarekomponenten, das in unterschiedlichsten Ausprägungen auftritt. In einer prozessintegrierten Umgebung übernimmt die Prozessmaschine die Aufgabe eines Mediators.

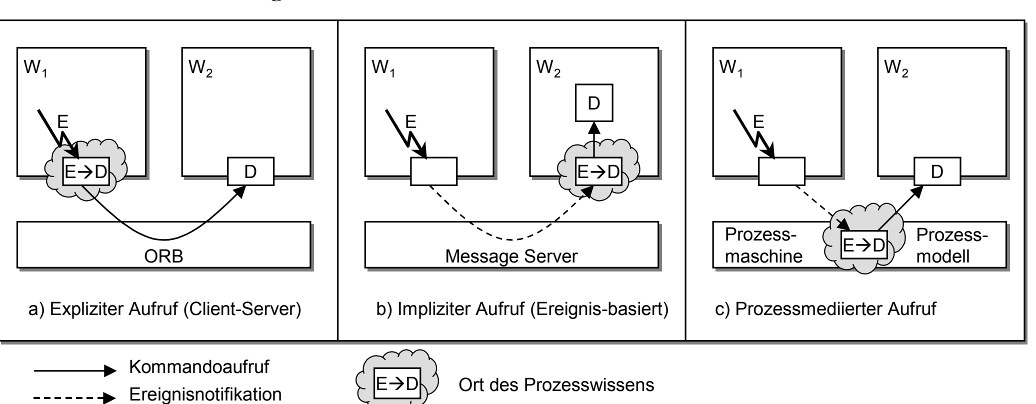


Abb. 11: *Verteilung des Prozesswissens bei unterschiedlichen Formen der Werkzeuginteraktion*

Abb. 11 illustriert schematisch die Verteilung des prozessrelevanten Interaktionswissens in den unterschiedlichen Ansätzen. Es wird angenommen, dass in einem Werkzeug W_1 ein prozessrelevantes Ereignis E auftritt, in dessen Folge in einem Werkzeug W_2 der Dienst D ausgeführt werden soll. Ein konkretes Beispiel eines solchen Ablaufs wäre die Speicherung eines Quelltextes (Ereignis E) in einem Editor (Werkzeug W_1) und die nachfolgende Übersetzung (Dienst D) in einem Compiler (Werkzeug W_2).

Im Falle einer traditionellen Client-Server-Architektur ruft das Werkzeug W_1 durch ein *explizites Kommando* direkt den Dienst D beim Werkzeug W_2 auf (Abb. 11, links). Dann liegt das Prozesswissen, dass bei Eintritt des Ereignisses E der Dienst D aufzurufen ist, bei Werkzeug W_1.

Im Fall eines rein ereignisbasierten Ansatzes löst Werkzeug W_1 den Dienst D in Werkzeug W_2 *implizit* aus, indem es eine Notifikation, dass E eingetreten ist, aussendet. Voraussetzung ist dann, dass das Werkzeug W_2 weiß, wie das Ereignis E zu behandeln ist. Im diesem Fall ist der Initiator des prozessrelevanten Ereignisses (W_1) zwar stärker von den anderen Werkzeugen entkoppelt. Das Prozesswissen über die „richtige" Reaktionen auf Ereignisnachrichten ist jedoch im Code der Nachrichtenschnittstelle des Werkzeugs W_2 versteckt (Abb. 11, Mitte).

Beide Ansätze führen zu offensichtlichen Problemen, wenn die Interaktion zwischen den Werkzeugen so abgeändert werden soll, dass bei Ereignis E der Dienst D durch eine anderen Dienst ersetzt werden soll oder zusätzliche Schritte als Reaktion auf E eingebaut werden sollen. Im konkreten Beispiel könnte beispielsweise dem Speichern eines Quelldatei erst das Einchecken in ein Versionsmanagementsystem folgen, bevor die Übersetzung gestartet wird.

Daraus lässt sich schlussfolgern, dass ein Werkzeug zum einen keine direkten Kommandos an andere Werkzeuge, sondern nur Notifikationen über das Auftreten von Ereignissen verschicken sollte. Zum anderen sollten Werkzeuge auf Empfangsseite keine Notifikationen behandeln müssen, sondern lediglich den direkten Aufruf von Diensten durch explizite Kommandos exportieren. In Umgebungen ohne explizite Kontrolle durch des Prozesses durch eine separate Komponente

bedeutet dies einen scheinbaren Widerspruch, denn wie soll Interaktion zwischen Werkzeugen stattfinden, wenn Sender nur Notifikationen aussenden und Empfänger nur Kommandos behandeln können?

Die Grundidee besteht darin, dass die Prozessmaschine als *Mediator* in den Nachrichtenverkehr zwischen den Werkzeugen geschaltet wird und die Abbildung von Notifikationen eines Werkzeugs auf Kommandos anderer Werkzeuge vornimmt sowie Abläufe auf Basis expliziter Prozessmodelle steuert (Abb. 11, rechts). Dadurch wird Prozesswissen aus den Werkzeugen in die Prozessmaschine bzw. die zugrunde liegenden Prozessmodelle verlagert. Wir sprechen bei dieser Anforderungen von der *prozessorientierten Mediation der Werkzeuginteraktionen*.

3.3.3.2 Bewertung existierender Ansätze

Zur Unterstützung von Werkzeuginteraktionen im Kontext von Entwurfsumgebungen existieren eine Reihe von Ansätzen, die sich grob in folgende Kategorien einordnen lassen [Sche93]:

- Direkte lokale und/oder entfernte Prozeduraufrufe;
- Object Request Broker-Architekturen;
- Message-Broadcasting-Architekturen;
- Mediator-Ansätze.

Im Folgenden stellen wir die grundlegenden Merkmale dieser Lösungsansätze vor und bewerten ihre Eignung für den Einsatz in einer prozessintegrierten Umgebung. Dabei folgen direkte Prozeduraufrufe und Object-Request-Broker-Ansätze dem Client-Server-Paradigma, während Message-Broadcasting-Architekturen dem ereignisbasierten Architekturstil zugerechnet werden können. Mediator-Ansätze kombinieren die Stärken der vorgenannten Architekturen und kommen unserer Zielvorstellung einer prozessorientierten Mediation der Werkzeuginteraktionen am nächsten.

Direkte Prozeduraufrufe

Die grundlegendste (und wegen ihrer Allgegenwärtigkeit oft gar nicht als solche wahrgenommene) Form der Kommunikation zwischen Softwarekomponenten stellen *Prozeduraufrufe* dar. Hierbei wird ein Werkzeugdienst als einzelne Prozedur aufgefasst, welche von anderen Werkzeugen oder Umgebungsdiensten aufgerufen werden kann. Dies setzt allerdings voraus, dass die gesamte Umgebung durch einen Betriebssystemprozess realisiert wird, was jedoch leicht zu monolithischen Softwarearchitekturen führt, die auch unter Performanz-Gesichtspunkten schnell an ihre Grenzen stoßen. Die Verwendung entfernter Prozeduraufrufe (*Remote Procedure Call, RPC*) [Schi92a; Schi92b] ermöglicht auch Aufrufbeziehungen über Betriebssystemprozessgrenzen hinweg, so dass eine Entwurfsumgebung aus unabhängigen, in einem Netzwerk verteilten Werkzeugen gebildet werden kann. Die Basis hierfür stellen Middleware-Infrastrukturen wie *Sun-RPC* oder *DCE-RPC* [Schi93] bereit. Diese Infrastrukturen zielen vor allem syntaktische und semantische Gleichbehandlung lokaler und entfernter Prozeduraufrufe ab (mit Ausnahme Zeiger-wertiger Parameter, die normalerweise bei entfernten Aufrufen nicht erlaubt sind, da sie ein „tiefes" Kopieren (*deep copy*) von Speicherstrukturen auf den entfernten Rechner bedingen würden). Als Grundproblem erweist sich jedoch bei

Remote Procedure Calls

lokalen ebenso wie bei entfernten Prozeduraufrufen die mangelnde Transparenz der *Bindung* zwischen den interagierenden Werkzeugen. Es lassen sich lediglich 1:1-Interaktionsbeziehungen auf niedrigem Abstraktionsniveau direkt abbilden. Hierbei muss das aufrufende Werkzeug den genauen Ort und die Schnittstelle des Diensterbringers kennen, so dass durch die entstehenden Abhängigkeiten eine nachträgliche Erweiterung oder ein Austausch von Werkzeugdiensten mit hohem Aufwand verbunden ist. Zudem ist das Ablaufwissen, dass zu einem bestimmten Zeitpunkt der Dienst eines anderen Werkzeugs zu aktivieren ist, im aufrufenden Werkzeug kodiert. So genannte *Multicast-RPCs* [BiJo87; WaZZ93] erlauben zwar 1:n-Aufrufbeziehungen (d.h. den gleichzeitigen Aufruf einer Prozedur bei einer *Gruppe* von Servern, die die betreffende Schnittstelle exportieren), ändern aber nichts an der grundsätzlichen Tatsache, dass Ablaufbeziehungen in den interagierenden Werkzeugen selbst verborgen sind.

Object Request Broker

Der Object-Request-Broker-Ansatz stellt eine objektorientierte Erweiterung des entfernten Prozeduraufrufs dar. Werkzeuge werden hier als (komplexe) Objekte aufgefasst, die über eine definierte Schnittstelle ihre Dienste zur Verfügung stellen. Gemäß dem objektorientierten Paradigma verfügt jedes (laufende) Werkzeug über eine eindeutige Objektidentität und einen eigenen Zustand, so dass sich anders als beim entfernten Prozeduraufruf mehrere Instanzen eines Werkzeugs unmittelbar unterscheiden lassen.

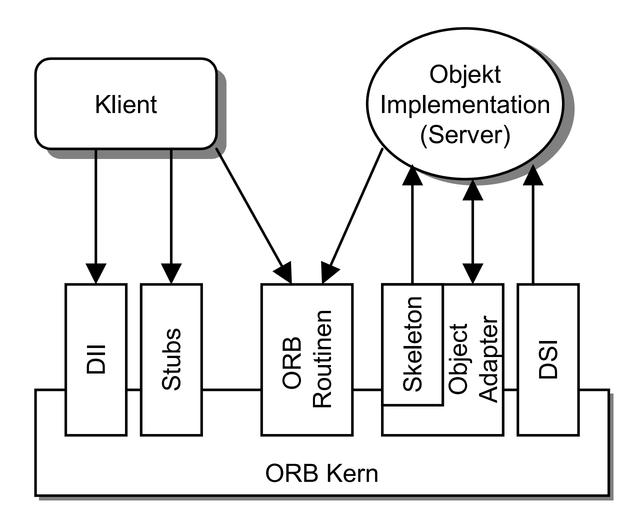


***Abb. 12**: CORBA-Architektur (vgl. [OMG#97; Grif98])*

Als typischen Vertreter des Broker-Ansatzes betrachten wir in Abb. 12 die *Common Object Request Broker Architecture* (CORBA), deren Spezifikation das Ergebnis der Bemühungen der OMG ist [OMG#97; Sieg96; Kräm97]. Konkurrierende Ansätze wie Microsofts DCOM [EdEd98; Tho*97] oder die Voyager-Technologie der Firma ObjectSpace folgen in den wesentlichen Elemente ihrer Architektur einem vergleichbarem Aufbau, auch wenn bei genauerem Hinsehen hinsichtlich des zugrunde liegenden Objektmodells und der zur Verfügung stehenden Zusatzdienste doch erhebliche Unterschiede existieren (für einen Vergleich zwischen DCOM und CORBA siehe z.B. [OrHE96; Grif98; Chu*98]).

In einer CORBA-basierten Umgebung kommunizieren Werkzeuge über den so genannten *Object Request Broker* (ORB) miteinander. Dazu verwendet das Klient-Objekt, d.h. das aufrufende Werkzeug, die bei der Übersetzung aus einer Schnittstellenspezifikation erzeugten Coderümpfe (*stubs*). Hier findet unter anderem die

Serialisierung der Parameter in einen über das Netzwerk verschickbaren Bytestrom, das so genannte *Marshalling*, statt. Der ORB ist für die transparente Weiterleitung von Operationsaufrufen innerhalb einer heterogenen, verteilten Systemumgebung zuständig. Seine Aufgaben umfassen unter anderen die Generierung und Auswertung von Objektreferenzen, die Auflösung von Operationsaufrufen (*method dispatch*), die Aktivierung und Deaktivierung von Objekten und die Registrierung des die Objekte implementierenden Codes. Serverseitig, d.h. beim diensterbringenden Werkzeug, werden die Aufrufe analog von den ebenfalls automatisch erzeugten *Skeletons* entgegen genommen und an die Implementierung des betreffenden Dienstes weitergereicht. In die eigentliche Kommunikationsanbindung an das Zielobjekt ist noch so genannter *(Basic) Object Adapter* (BOA) zwischengeschaltet. Dieser kann ausgetauscht werden und spezielle Funktionalitäten anbieten, um zum Beispiel Serverobjekte nicht als eigene Betriebssystemprozesse, sondern als Datenbankobjekte zu realisieren.

Dynamic Invocation Interface

Alternativ zur Anbindung über zur *Übersetzungszeit* bekannte Stubs und Skeletons kann der Operationsaufruf auch über das *Dynamic Invocation Interface* (DII) erfolgen, welches dem Klienten die Angabe der Operation und der Parameter *zur Laufzeit* erlaubt. Die analoge Funktionalität stellt auf Serverseite das *Dynamic Skeleton Interface* (DSI) zur Verfügung. Zusammen mit Namensdiensten [Sieg96] und Tradingdiensten [PoSW96; AT&T96] kann so das Auffinden und die Aktivierung von diensterbringenden Werkzeugen für das aufrufende Werkzeug weitgehend transparent gestaltet werden.

Abläufe als Geflecht von Punkt-zu-Punkt-Beziehungen zwischen verschiedenen Werkzeugobjekten

Zu einer *prozessadaptablen* Komposition von Werkzeugdiensten zu komplexeren Ablauffragmenten trägt die durch DII/DIS, Namens- und Tradingdienste gewonnene Flexibilität allerdings nur wenig bei, da diese Mechanismen im Wesentlichen die *Orttransparenz* der Bindung zwischen Werkzeugen begünstigen, d.h. die Werkzeuge müssen nicht statisch zur Übersetzungszeit ihre Kommunikationspartner kennen. Prozessrelevantes Ablaufwissen drückt sich jedoch – genau wie beim direkten Prozeduraufruf – immer noch durch ein Geflecht von *Punkt-zu-Punkt*-Beziehungen zwischen den beteiligten Werkzeug-Objekten aus [Brow93; GaIl90]. Diese Interaktionsabfolgen sind nicht explizit, d.h. unabhängig von den beteiligten Werkzeugen, definiert, sondern manifestieren sich lediglich implizit in den vom aufrufenden Werkzeug initiierten Bindungen und sind als Codeabschnitte über die Objektmethoden der einzelnen Werkzeuge verteilt [PaSa96].

Message Broadcasting

Unterscheidung zwischen Kommandos und Nachrichten

In so genannten Message-Broadcasting-Architekturen kommunizieren Werkzeuge über den Austausch von Nachrichten. Zentraler Bestandteil von Message-Server-Architekturen ist ein so genannter *Broadcast Message Server* (BMS), der für die Entgegennahme, Weiterleitung und Verteilung von Nachrichten sowie die Verwaltung der laufenden Werkzeuginstanzen zuständig ist. Nachrichten werden unterschieden in *Kommandos* und *Notifikationen*. Kommandos werden an ein Werkzeug oder eine Gruppe von Werkzeugen geschickt werden, um dort eine vom aufrufenden Werkzeug festgelegte Operation zu veranlassen. Kommandos sind in der Regel synchrone Nachrichten (d.h. der Versender der Nachricht wartet auf eine Antwort) und entsprechen im Wesentlichen den in den vorgenannten Ansätzen angesprochenen entfernten Prozedur- bzw. Methodenaufrufen mit dem Unterschied, dass hier auf natürliche Weise auch 1:n-Beziehungen zwischen aufrufendem und empfangenden Werkzeugen realisiert werden können.

Die eigentliche Besonderheit des Message-Broadcasting-Ansatzes stellt jedoch das Konzept der *Notifikationen* dar, mit dessen Hilfe das Prinzip des *impliziten Operationsaufrufs* [SuNo92; GaIl90] realisiert werden kann. Werkzeuge melden *asynchron* bestimmte Ereignisse oder Änderungen ihres Zustands an eine für sie *anonyme* Menge von Interessenten, welche dann entsprechend auf diese Meldungen reagieren können. Die Ereignisse werden als Nachrichten von einem ereignismeldenden Werkzeug an den BMS geschickt und von dort an interessierte Werkzeuge weitergeleitet. Um über das Eintreffen eines bestimmten Ereignisses informiert zu werden, muss sich ein Werkzeug vorher beim BMS registriert und sein Interesse durch Angabe entsprechender Nachrichtenmuster bekundet haben[8]. Im Gegensatz zu den vorher betrachteten Ansätzen werden Werkzeugdienste also nicht direkt aufgerufen, sondern implizit durch Ereignismeldungen ausgelöst. Das Wissen, wie auf ein bestimmtes Ereignis zu reagieren ist, liegt bei den Empfängern.

Impliziter Operationsaufruf durch Notifikation

Da der Produzent einer Nachricht von den Empfängern entkoppelt ist, besteht ein großer Vorteil des Ansatz in der einfachen Austauschbarkeit von Werkzeugen und der Erweiterbarkeit um neue Werkzeuge, die in den Nachrichtenstrom eingeklinkt werden können, ohne dass dazu Änderungen an den existierenden Werkzeugen erforderlich sind. Voraussetzung ist allerdings eine vorherige Einigung auf einen Satz von Standardnachrichten, die von allen zu integrierenden Werkzeugen verstanden werden. Da Nachrichten in der Regel Parameter beinhalten, die auf Produktdaten verweisen, berührt die Integration über einheitliche Nachrichtenprotokolle auch den Bereich der Datenintegration (vgl. Abschnitt 3.3.2).

Erweiterbarkeit

Mittlerweile haben Message-Broadcasting-Mechanismen eine recht weite Verbreitung in einer Reihe von Entwurfsumgebungen gefunden. Die unterschiedlichen Ansätze unterscheiden sich vor allem durch den Informationsgehalt und Standardisierungsgrad der Nachrichten sowie durch zusätzliche Hilfsmittel zur Einbindung von Fremdwerkzeugen. Als Mutter aller Message-Broadcasting-Ansätze gilt die im akademischen Umfeld entstandene *FIELD*[9]-Umgebung, die von S. Reiss an der Brown University entwickelt wurde [Reis90; Reis90a]. Hierbei handelt es sich um eine Programmierumgebung, in der vor allem einfache Kommandozeilen-orientierte Werkzeuge für das Programmieren-im-Kleinen integriert worden sind. Entsprechend simpel ist das Format von Nachrichten und Nachrichtenmustern gehalten.

FIELD

Das *Softbench-Rahmenwerk* von Hewlett-Packard ist ein kommerzieller Nachfolger der Field-Umgebung, der eine Reihe interessanter Erweiterungen aufweist [Caga90; BrEM92; Bro*94; FrWa93]:

SoftBench

- Werkzeuge werden vordefinierten *Werkzeugklassen* (z.B. Editor, Compiler, Debugger) zugeordnet, für die jeweils ein *Werkzeugprotokoll*, d.h. ein Satz von Standardnachrichten spezifiziert wurde, die jedes Werkzeug dieser Klasse verstehen sollte. Zum Beispiel sollten alle Texteditoren auf die Nachricht, dass in einem Textfile in einer bestimmten Zeile ein Fehler aufgetreten ist, entsprechend reagieren können, indem sie die betreffende Textdatei öffnen und an die fragliche Stelle springen. Auf diese Art und

[8] Da im Allgemeinen nicht alle Werkzeugen an allen Ereignissen interessiert sind, wäre übrigens *selektives Musticasting* eine korrektere Bezeichnung für diesen Mechanismus zur Nachrichtenverteilung als Broadcasting.

[9] FIELD: Friendly Integrated Environment for Learning and Developing

Weise ist ein Mindestmaß an semantischem Verständnis zwischen den Werkzeugen gewährleistet und Werkzeuge ein und der selben Klasse können untereinander ausgetauscht werden.

- Ein spezielles Softbench-Werkzeug, der *Encapsulator*, erleichtert die Einbindung von Fremdwerkzeugen in Softbench, indem es die Generierung von Wrapper-Schnittstellen für das Versenden und Empfangen von Nachrichten unterstützt. Dieser Wrapper interpretiert die Ein- und Ausgaben eines Werkzeugs, wandelt sie in entsprechende Nachrichten um und versieht das Werkzeug bei Bedarf zusätzlich mit einer interaktiven Benutzungsschnittstelle des zugrunde liegenden Fenstersystems. Ohne massiven Eingriff in das zu integrierende Werkzeug funktioniert dieser Ansatz allerdings nur bei einfachen, Batch-artigen Kommandozeilen-Werkzeugen zufriedenstellend.
- Ein *Execution Manager* ist für das transparente Auffinden und Aktivieren einer passenden Werkzeuginstanz zuständig, wenn ein Werkzeug einen bestimmten Dienst angefordert hat.
- Es werden neben Anforderungsnachrichten (*request messages*) Erfolgsmeldungen (*success notifications*) und Fehlermeldungen (*failure messages*) unterschieden.

ToolTalk

Ein weiterer Vertreter des Message-Broadcasting-Ansatzes ist Sun's *ToolTalk* [Fran91; Suns93]. Hier wurde wieder die Grundidee eines zentralen Message Servers übernommen. Allerdings weist auch ToolTalk einige Besonderheiten auf:

- Im Gegensatz zur rein prozeduralen Sicht in FIELD und SoftBench gibt ToolTalk einen Migrationsschritt in Richtung Objektorientierung vor. Werkzeuge lassen sich als so genannte *object descriptions* repräsentieren, die untereinander über Nachrichten kommunizieren. Durch Anordnung der *object descriptions* innerhalb einer Vererbungshierarchie lassen sich die Nachrichtenschnittstellen und Interessenprofile zwischen Werkzeugklassen vererben.
- In ToolTalk existiert ein S*ession*-Konzept, wobei jeweils ein ToolTalk-Server mit mehreren Werkzeugen einer Session zugeordnet ist. Pro Benutzer können eine oder mehrere Sessions aktiv sein. Je nach Nachrichtenspezifikation erfolgt die Nachrichtenverteilung lokal oder Session-übergreifend (siehe unten), wodurch auch die Kollaboration mehrerer Benutzer unterstützt werden kann.
- Das Nachrichtenformat ist in ToolTalk wesentlich reichhaltiger als in FIELD oder SoftBench. Die Attribute, aus denen sich eine Nachricht zusammensetzt, umfassen:
  - *Address*: Adressat einer Nachricht kann entweder eine Prozedur, ein (Betriebssystem-)Prozess, ein Objekt oder ein Objekttyp sein.
  - *Class*: es werden zwei Klassen von Nachrichten unterschieden: Anforderungen (*request*) und Notifikationen (*notice*).
  - *Operation:* über dieses Attribut wird die aufzurufende Operation (bei Anforderungen) oder das aufgetretene Ereignis (bei Notifikationen) spezifiziert.

- *Arguments:* hiermit wird eine Liste von Parametern zu einer Anforderungs- oder Notifikationsnachricht spezifiziert
- *Scope:* Mit Hilfe dieses Attributs wird ein Gültigkeitsbereich für die Verteilung einer Nachricht definiert. Mögliche Werte sind *session* (alle Werkzeug innerhalb der lokalen Sitzung), *file* (alle Werkzeuge, die eine bestimmte Datei bearbeiten, d.h. auch Sitzungs-übergreifend), *both* (Vereinigungsmenge aus *session* und *file*) und *file-session* (Schnittmenge aus *session* und *file*).
- *Disposal*: Dieses Attribut gibt an, was ToolTalk für den Fall, dass kein passender Empfänger einer Nachricht gefunden wird, tun soll.
- *State:* Über dieses Attribut wird dem Absender einer Nachricht mitgeteilt, ob ein Empfänger gefunden wurde oder nicht.

Weitere Message-Broadcasting-Ansätze

Neben FIELD, Softbench und ToolTalk existieren noch eine Reihe weiterer Ansätze und kommerzieller Produkte, die auf dem Message Broadcasting-Prinzip basieren oder zumindest Architekturen gemäß diesem Paradigma ermöglichen. Als kommerzielle Ansätze sind ist die *FUSE*-Umgebung von Digital [DEC#93] und die *SDE Workbench/6000* von IBM zu nennen.

JavaBeans

JavaBeans [Engl97], das Komponentenmodell von Sun auf Basis der Programmiersprache Java [Flan00], definiert einen Ereignismechanismus, mit dessen Hilfe sich JavaBeans-Komponenten wechselseitig über Zustandsänderungen informieren können. Das zugrunde liegende Ereignismodell basiert auf einer *EventSource* (dem ereignismeldenden Objekt), das Ereignisobjekte an ein oder mehrere *Event-Listener-Objekte* sendet, die eine definierte Schnittstelle implementieren und sich vorher beim *EventSource-Objekt* registriert haben müssen. Zur Entkopplung von Source- und Listenerobjekt sowie zur Anpassung inkompatibler Schnittstellen können *Eventadapter* zwischengeschaltet werden, die als Filter, Verteiler und Zwischenspeicher dienen und somit eine Rolle ähnlich der eines Message-Servers in Message-Broadcasting-Architekturen übernehmen. Die Eventadapter müssen jedoch (mit Unterstützung durch eine entsprechende Entwicklungsumgebung, z.B. durch Sun's JavaStudio) explizit ausprogrammiert werden. Zudem fehlt der Java-Beans-Architektur eine „offizielle" Verteilungsmöglichkeit, d.h., JavaBeans können nur lokal innerhalb *einer* virtuellen Java-Maschine kommunizieren [Grif98].

Eventadapter

JEDI, ESF Software Bus und Polylith

JEDI ist eine weitere Java-basierte Infrastruktur zur Ereignisverteilung, die innerhalb der prozesszentrierten Umgebung *OPSS* eingesetzt wird [CuDF98]. Der Software-Bus des *Eureka Software Factory-Projekts* (ESF) [FeNO92] fällt ebenfalls in die Kategorie der Message-Broadcasting-Architekturen, wobei hier als Besonderheit in Anlehnung an das Model-View-Controller-Paradigma von Smalltalk [Gold84] eine strenge Trennung zwischen so genannten Dienstkomponenten (*service components*) und Interaktionskomponenten (*user interaction components*) propagiert wird. Als weiteres System macht *Polylith* [Purt94] Gebrauch von der Metapher eines Software-Busses, in den Software-Komponenten in Analogie zu Hardware-Komponenten nach Bedarf hineingesteckt und wieder herausgenommen werden können.

Ereignisdienste in CORBA und DCOM

Auch in CORBA wurde, als Teil der so genannten *Common Object Services* (COS), das Konzept der *event channels* spezifiziert [OMG#97b] und in einigen Implementationen realisiert [EiMC97]. In DCOM steht mit den *event sinks* bzw. den *connectable objects* ein ähnlicher Mechanismus zur Ereignisverteilung zur Verfü-

gung [EdEd98]. Mit diesen Diensten lassen sich auch in CORBA und DCOM Architekturen nach dem Message-Broadcasting-Prinzip emulieren – allerdings nur auf vergleichsweise niedrigem logischen Niveau [CuDF98; PaSa96].

Für eine weiterführende, detaillierte Gegenüberstellung verschiedener ereignisbasierter Integrationsansätze sei der Leser auf den *EBI*[10]-Klassifikationsrahmen von Barrett et. al. [BCTW96] verwiesen. Abschließend weisen wir noch darauf hin, dass sich das hinter dem Message-Broadcasting-Ansatz steckenden Entwurfsprinzip des *impliziten Aufrufs* nicht nur bei der Integration von Entwurfsumgebungen, sondern auch in ganz anderen Kontexten (z.B. aktive Datenbanken [DiGa00], inkrementelle Attributauswertung [TeRe88], automatische Typumwandlung [ClOs90], Dämonen auf Betriebssystemebene [HaNo86] oder Blackboard-Architekturen in KI-Systemen [JaDB89]) Anwendung findet.

Mediator-Ansätze

Die oben skizzierten Ansätze zum *impliziten Aufruf* zeigen zwar eine ganz deutliche Weiterentwicklung des Message-Broadcasting-Konzepts hinsichtlich der zugrunde liegenden Nachrichten- und Werkzeugmodelle (FIELD: unstrukturierte Nachrichten, SoftBench: standardisierte Protokolle für vordefinierte Werkzeugklassen, ToolTalk: objektorientierte Sichtweise auf Nachrichten und Werkzeuge), ohne jedoch etwas am grundsätzlichen Integrationsprinzip zu ändern: während der Sender einer Nachricht durch die Indirektion über den Message Server von den möglichen Kommunikationspartnern entkoppelt ist, liegt das Wissen über die „richtige" Reaktion auf eine Nachricht, d.h. die Umsetzung einer Nachricht in einen effektiven Dienstaufruf, beim Empfänger-Werkzeug. Dies erschwert die Anpassbarkeit der dadurch inhärent in den Werkzeugen festgelegten Abläufe erheblich.

Trennung von Verarbeitung und Koordination

In Hinblick auf die geforderte prozessorientierte Adaptabilität der Interaktionsbeziehungen zwischen Werkzeugen bzw. ihren Diensten führt dies zu der Forderung, dass die Interaktionspartner noch stärker voneinander entkoppelt werden müssen. Die – auch in ganz anderen Zusammenhängen [MaCr94] anzutreffende – Lösungsidee besteht in einer konsequenten Trennung und orthogonalen Behandlung der Aspekte *Verarbeitung* (Werkzeuge, deren Dienste und Ereignisse) und *Koordination* (Organisation des Verhaltens einer Gruppe von Werkzeugen durch Verwaltung und Steuerung ihrer Dienste). Die bei der Komposition von Werkzeugdiensten zu komplexeren Abläufen auftretenden Interaktionsbeziehungen sind nicht mehr Teil der Werkzeuge selbst, sondern werden in eine eigene Komponente verlagert.

Mediation als Entwurfsprinzip

Die softwaretechnische Abstraktion dieses Entwurfsprinzips findet sich z.B. in Sulliver's *Mediator-Methode* [SuNo92; SuKN96]. Ein Mediator wird hier als ein Konstrukt verstanden, das die Interaktionen zwischen Softwarekomponenten kapselt, wodurch diese voneinander entkoppelt werden und die Semantik der Interaktionen vollständig durch den Mediator definiert wird. Auch in der objektorientierten Entwurfsmethodik begegnet uns die Grundidee des *entkoppelten und mediierten Aufrufs* immer wieder in Form unterschiedlicher *Entwurfsmuster*, die eine lose Kopplung zwischen Softwarekomponenten mit dem Ziel der Variation be-

[10] EBI: Event-Based software Integration

stimmter Aspekte erlauben [GHJV95; CoSc95; Bus*96]. So zielt das *Mediator*-Muster auf die Variation der Komponenteninteraktion, indem die Koordination des Zusammenspiels zwischen Komponenten in eine eigenständige Komponente verlagert wird. In eine ähnliche Richtung geht das *Observer*-Muster, das eine Variation des Veränderungsverhaltens zwischen beobachteten und beobachtenden Komponenten gemäß dem in Smalltalk propagierten Model-View-Controller-Paradigma [Gold84] ermöglicht.

Entwurfsmuster Mediator *und* Observer

Im Kontext der Werkzeugintegration in Entwurfsumgebungen findet sich eine *echte Erweiterung* des Message-Broadcasting-Ansatzes in Richtung eines Meditiator-Konzepts bereits in *Forest* [GaIl90], einem frühen Nachfolger der FIELD-Umgebung. Anders als bei FIELD werden ankommende Nachrichten (insbesondere Notifikationen) nicht einfach gemäß der Registrierung der Nachrichtenmuster an interessierte Werkzeuge weitergeleitet. Stattdessen wird die Verteilung durch den Message Server mit Hilfe so genannter *policy descriptions* gesteuert. Diese Regelsätze, die vom Umgebungsadministrator oder auch vom Benutzer dynamisch geändert werden können, haben die Form von *Event-Condition-Action*-(ECA)-Klauseln. Der *Ereignisteil* korrespondiert mit ausgesendeten Nachrichten der Werkzeuge. Der *Bedingungsteils* ist ein Boolescher Ausdruck, der sich aus aussagenlogischen Junktoren ($\wedge$, $\vee$, $\Rightarrow$, $\neg$), numerischen Ausdrücken, vordefinierten Systemfunktionen und benutzerdefinierten Statusvariablen zusammensetzt. Der *Aktionsteil* kann drei mögliche Formen annehmen. Im Normalfall besteht die Aktion im Versenden einer Nachricht an ein Empfänger-Werkzeug. Diese Nachricht ist dabei in der Regel nicht identisch mit der Nachricht, die die Auswertung der *policy description* verursacht hat. Ein typischer Anwendungsfall ist hier, dass eine Notifikation eines Werkzeugs auf ein Kommando eines anderen Werkzeugs abgebildet wird. Der Vorteil liegt also darin, dass ein Werkzeug (auf Empfängerseite) nur noch über eine Kommando-Schnittstelle verfügen muss, während die Reaktion eines Werkzeugs auf ein Ereignis (Notifikation) außerhalb des Werkzeugs durch die *policy description* bestimmt wird. Mit Hilfe einer `null`-Aktion wird eine Nachricht komplett ignoriert. Des weiteren können neue Nachrichten an den Message Server selbst geschickt werden, wodurch die Auswertung weiterer *policy descriptions* initiiert wird.

Forest: Erweiterung des Message Servers um policy descriptions

Die *policy descriptions* tragen also im Vergleich zu Message-Broadcasting-Architekturen stärker dazu bei, Prozesswissen aus den Werkzeug zu ziehen und auf einer externen, adaptablen Ebene zu repräsentieren. Der Message Server rückt dadurch von der Rolle eines passiven Nachrichtenverteilers in die eines aktiven Mediators, der die Interaktionsbeziehungen zwischen Werkzeugen koordiniert. Die Mediation des Forest-Message-Servers kann man jedoch auf Grund der eingeschränkten Ausdrucksmächtigkeit der *policy description* noch nicht als prozessorientiert bezeichnen. Insbesondere lassen sich keine mehrschrittigen Abläufe steuern, da der Message Server bei der Auswertung der *policy descriptions* immer nur die aktuell eingetroffene Nachricht berücksichtigt und keinen Ablaufkontext aufbewahrt.

Geringe Ausdrucksstärke der policy descriptions

Einen von der Grundidee mit Forest vergleichbaren Ansatz verfolgt die *ToolBus*-Architektur [BeKl96]. Wie in Forest kommunizieren die Werkzeuge auch hier über einen Message-Server, der hier als *ToolBus* bezeichnet wird. Die Interaktion zwischen Werkzeugen wird hier durch so genannte *T-Skripte* gesteuert. T-Skripte aggregieren Basisfunktionalitäten von Werkzeugen zu höherwertigen Diensten, indem sie eine Folge von Kommunikationsprimitiven für das Empfangen und

T-Skripte in ToolBus

Senden von Nachrichten mit Hilfe der Operatoren Sequenz, Alternative und Iteration verketten. Die T-Skripte werden von den Autoren selbst als *prozessorientiert* bezeichnet, sind jedoch auf einem sehr niedrigen, technischen Abstraktionsniveau angesiedelt.

Prozessorientierte Mediation in prozesszentrierte Umgebungen

Einen Schritt weiter in Richtung einer *prozessorientierten* Mediation der Werkzeuginteraktionen geht die Kombination von Message-Broadcasting-Architekturen mit Prozessmodellen und -maschinen, wie sie in einer Reihe von prozesszentrierten Entwicklungsumgebungen (z.B. *SPADE* [BBFL94; BFGL94; BaDF96], *Merlin* [JPSW94], *ProcessWeaver* [Fern93], SMART [Gar*94]) anzutreffen ist. Die wesentlichen Merkmale der Werkzeugintegration innerhalb dieser Klasse von Systemen [SFGJ99] lassen sich stellvertretend gut am Beispiel der Architektur der SPADE-Umgebung illustrieren (siehe Abb. 13); die anderen genannten Umgebungen verfolgen eine analoge Integrationsstrategie, wobei die zugrunde liegenden Prozessmodellierungssprachen allerdings stark differieren.

SPADE

In SPADE besteht das *user interaction environment*, also die Durchführungsdomäne, aus der DEC Fuse-Umgebung, einer Sammlung von Programmierwerkzeugen (Editoren, Compiler, Debugger etc.), welche über den DEC Fuse Message Server (s.o.) nach dem Prinzip des Message-Broadcasting kontrollintegriert sind (Abb. 13, oben). In der Leitdomäne, hier *process enactment environment* genannt, interpretieren (hierarchisch angeordnete) Prozessmaschinen die Prozessmodelle, die in der Petrinetz-basierten Sprache SLANG [BaFG93] formuliert sind (Abb. 13, unten). Die Verbindung zwischen den beiden Domänen wird über eine spezielle Komponente, das SPADE Communication Interface (SCI), hergestellt (Abb. 13, Mitte). Aus Sicht des Fuse Message Servers stellt das SCI lediglich ein weiteres Werkzeug dar, das ebenso wie die originären Fuse-Werkzeuge Nachrichten von anderen Werkzeugen empfangen bzw. Nachrichten an diese verschicken kann.

Die Aufgabe des SCI besteht nun zum einen darin, Nachrichten, die von den Werkzeugen der Durchführungsdomäne generiert wurden, abzufangen und in den aktuell interpretierten SLANG-Netzen als Token in spezielle Stellen, die so genannten *user interface places*, abzubilden. Hierdurch ist es möglich, Ereignisse der Durchführungsumgebung in die laufende Prozessmodellinterpretation einfließen zu lassen. Umgekehrt lassen sich aus der Prozessmodellinterpretation heraus Werkzeugdienste in der Benutzerumgebung aktivieren. Hierzu ist in SLANG das Konzept der *black transitions* vorgesehen. Beim Schalten einer solchen Transition wird über das SCI eine in der Transition spezifizierte Kommandonachricht an den Message Server abgesetzt und von dort an das betreffende Werkzeug weitergeleitet.

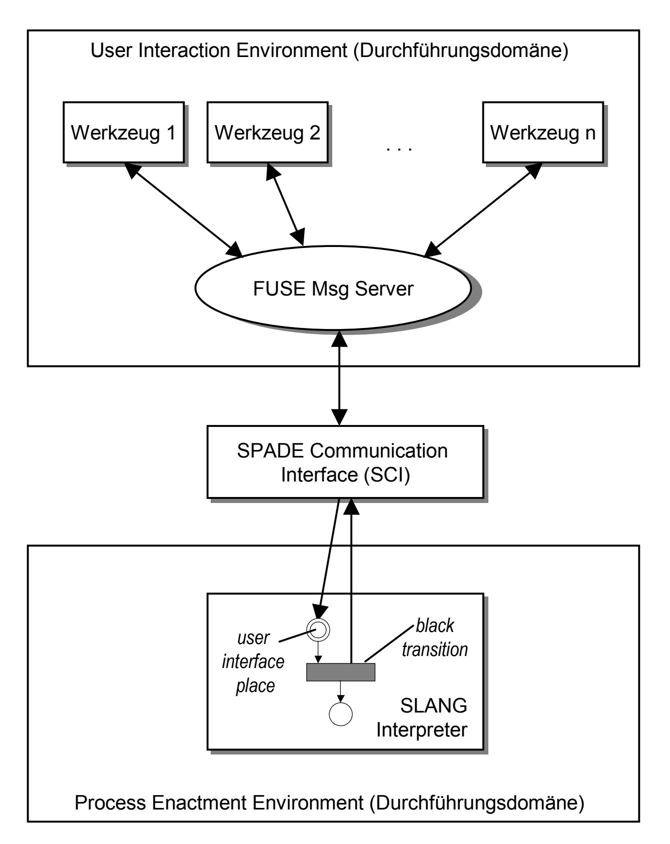


***Abb. 13**: Kopplung von Message-Broadcasting und Prozessmaschinen am Beispiel von SPADE [BFGL94; BaDF96]*

Konflikte zwischen Prozessmodellen und hartkodierten Werkzeugreaktionen

Ein gravierendes Problem ergibt sich jedoch aus der Tatsache, dass die eingesetzten Werkzeuge der DEC Fuse-Umgebung nicht eigens für den Einsatz in der prozesszentrierten Umgebung konzipiert wurden und bereits *eigene Reaktionen* auf bestimmte Nachrichten realisieren. Wenn Nachrichten sowohl an die Werkzeuge, als auch an die Prozessmaschine (via SCI) geleitet werden, besteht die Gefahr, dass die in einem Werkzeug einkodierten und die im Prozessmodell vorgesehenen Reaktionen auf eine Nachricht miteinander *konfligieren*. Ob und wie diese Konflikte, die eine Ausprägung des so genannten „process-in-the-tool"-Syndroms [Mont94; AnGr94] darstellen, in SPADE aufgelöst werden, ist der verfügbaren Literatur leider nicht zu entnehmen. Ein entscheidender Unterschied zum Forest-Ansatz besteht also darin, dass der Message Server nicht direkt um prozessorientierte „Politiken" angereichert werden, sondern dass die Prozessmaschine mit den Prozessmodell *neben* dem Message Server gleichberechtigt mit den Werkzeugen angesiedelt ist. Die Mediation der Werkzeuginteraktion ist also indirekt und gelingt daher nur unvollständig.

Koordinations-, Architekturbeschreibungs- und Skriptsprachen

Ansätze aus der komponentenorientierten Software-Entwicklung

Abschließend betrachten wir noch einige Ansätze der *komponentenorientierten Software-Entwicklung* [Grif98; Szyp98; NiDa95], zu denen sich im Zusammenhang mit der Mediation von Werkzeugbeziehungen eine Reihe von offensichtlichen Querbezügen ergeben, da (prozessorientierte) Werkzeugintegration ja auch als Spezialfall des allgemeineren Problems der Softwarekomposition verstanden werden kann [Estu99]. Interessant sind hier vor allem Konfigurations- und Kompositionstechniken, die die anwendungsspezifische Anpassung und das „Zusammenkleben" (*glueing*) von Softwarekomponenten innerhalb eines größeren Anwendungskontexts zum Ziel haben, sowie die damit verwandten Architekturbeschreibungs- und Skriptsprachen.

MediatorService

Der in [PaSa96; PaSE97; Satt97] vorgestellte *MediatorService* ist ein speziell auf die Werkzeugintegration in ingenieurwissenschaftlichen Entwurfsumgebungen zugeschnittener Dienst zur Definition, Verwaltung und Steuerung von Komponentenbeziehung in einem CORBA-basierten Systemumfeld. Die Komposition von Werkzeugdiensten wird mit Hilfe der *Tool Interconnection Language* (TIL), einer Erweiterung der CORBA *Interface Definition Language* (IDL), spezifiziert.

Synchronizer

Das in [Frol93; FrAg96] vorgestellte *Synchronizer*-Konzept ist ein Constraint-basiertes Sprachmittel für die Multiobjekt-Koordination. Das Haupteinsatzgebiet dieses Ansatzes liegt in der Zugriffssteuerung und –kontrolle von Klienten auf einen *gemeinsamen* Server in Hinblick auf die Atomarität, die temporale Ordnung und den gegenseitigen Ausschluss multipler Aufrufe an den Server.

Gluonen

In [Pint93] werden mit den so genannten *Gluonen* spezielle Objekte zur Kommunikationsvermittlung zwischen Objekten eingeführt. Ein Gluon kapselt die Interaktion zwischen einem Client-Objekt und einem Server-Objekt und kann insbesondere Anpassungen am Interaktionsprotokoll vornehmen. Da ein Gluon selbst wieder ein Objekt darstellt, kann es seinen Zustand speichern und so die Interaktionshistorie bei der Kommunikationsvermittlung einbeziehen. Die Definition von Gluonen findet im Gegensatz zu den vorgenannten Ansätzen nicht auf einer Spezifikationsebene, sondern auf der programmiersprachlichen Ebene statt. Gluonen können in einer Hierarchie angeordnet werden, die die Wiederverwendung und Spezialisierung von Interaktionsprotokollen unterstützt. Als Beispiel wird in [Pint93] der Einsatz von Gluonen innerhalb eines Frameworks für Finanzanwendungen angeführt.

Architekturbeschreibungssprachen

Aus dem Bereich der Software-Architekturmodellierung [Nagl90; ShGa96; BaCK98; Dono99] stammen eine Reihe von Architekturbeschreibungssprachen, so genannte *ADLs*[11] [Clem96; MeTa97; Vest93; KrMa97], deren Ursprung meist in der Entwicklung verteilter Systeme liegt, wo sich ebenfalls die Forderung nach unabhängigen Komponenten mit wohldefinierten Interaktionsbeziehungen ergibt. Beispiele sind u.a. *MetaH* [Vest96], *Rapide* [Luc*95], *Wright* [Alla97], *Darwin* [MDEK95], *Olan* [BAKR95a], *Polylith/MIL* [Purt94] und *Focus* [MüSc96]. Charakteristisch für diese Sprachen ist die separate Beschreibung von Komponenten, Konnektoren und daraus gebildeten Konfigurationen. Die meisten dieser Sprachen sind formal fundiert; so basiert die Semantik von Darwin auf dem π-*Kalkül* von Milner [Miln89], in Wright wird eine Untermenge der *Communicating Sequential Processes* (CSP) von Hoare [Hoar85] zur Spezifikation des Interaktionsverhaltens von Konnektoren verwendet.

Gemeinsames Merkmal der genannten Modelle ist die *explizite Repräsentation* der Interaktion einer Gruppe von Komponenten. Die Zielrichtung der eingesetzten Beschreibungstechniken ist allerdings etwas anders gelagert als in unserem Fall. Es steht weniger die *Ablaufsteuerung* zwischen Komponenten im Vordergrund, als vielmehr der Ausgleich von Schnittstellen- und Protokollinhomogenitäten zwischen verschiedenen Komponenten.

Interzeption

Dies gilt auch für *Interzeptoransätze*, die beim Übergang zwischen verschiedenen Komponentenarchitekturen (z.B. von JavaBeans nach ActiveX oder von CORBA nach DCOM) die zu verknüpfenden Komponenten von Aufrufanpassungen und

[11] ADL: Architecture Description Language

Typtransformationen befreien und semantische Unterschiede zwischen den Komponentenarchitekturen (z.B. unterschiedliche Interpretationen von Objektidentifikatoren, Bindungsinformationen, Sicherheitsschlüsseln, Fehlern etc.) kompensieren [Grif98]. Die Grundidee ist hier, die für eine heterogene Komponentenbindung und –interaktion erforderlichen Interpretations- und Transformationsvorgänge in einer eigenen Infrastrukturkomponente, dem *Interzeptor*, zu kapseln. Beispiele für solche Interzeptor-Komponenten sind die von JavaSoft angebotene *ActiveX-Bridge* [Java98], die die Nutzung von JavaBeans als ActiveX-Controls innerhalb von Microsofts Komponentenmodell COM ermöglicht, oder die COM/CORBA-Bridge *OrbixCOMet* [Iona00] von Iona Technologies.

Skriptsprachen

Während sich der Begriff der Koordinations- und Konfigurationssprachen vornehmlich in der wissenschaftlichen Literatur findet, ist in der praktischen Anwendung meist von *Skriptsprachen* die Rede. Hier hat in den vergangenen Jahren Microsofts *Visual Basic* (VB) und die speziell auf die Steuerung von Applikationsprogrammen zugeschnittene Variante *Visual Basic for Applications* (VBA) große Popularität erlangt und sich als Marktführer im Windows-Umfeld etabliert. Der Erfolg von VB/VBA bzw. seiner Laufzeitumgebung ist zu einem großen Teil darauf zurückzuführen, dass es den Umgang mit Microsofts *Automation*-Technologie[12] für die Fernsteuerung von komponentenbasierten Applikationen im Vergleich zu anderen Sprachen, z.B. C++, drastisch vereinfacht und die Komplexität des zugrunde liegenden *Component Object Model* (COM) weitgehend verbirgt. Obwohl von Microsoft mittlerweile als vollwertige Sprache für beliebige Anwendungen propagiert, eignet sich Visual Basic besonders für die Automation und Integration von Aktivitäten innerhalb der Microsoft Office-Familie. Im Zusammenhang mit Internettechniken und der Programmierung von Web-Browsern spielen *VB Script* [Lamp99], eine abgespeckte Version von VBA, und Netscapes Pendant *JavaScript* [Mint97] eine wichtige Rolle. Ein weiterer wichtiger Vertreter der Skriptsprachen, um den es in letzter Zeit allerdings etwas still geworden ist, ist die Tool Command Language *Tcl* [Oust94]. Die historischen Wurzeln dieser Sprache liegen in dem Wunsch, einzelne, in der Sprache C geschriebene Programme zu fertigen Applikationen zusammenfassen zu können. Ihre eigentliche Popularität verdankt die Tcl der mitgelieferten Erweiterung *Tk*, einem grafischen *Toolkit* zur schnellen und plattformübergreifenden Erstellung von Benutzeroberflächen.

Visual Basic

VB Script, JavaScript und Tcl/Tk

Gemeinsam ist allen Skriptsprachen, dass sie von den angebotenen Ablaufstrukturen herkömmlichen prozeduralen Programmiersprachen ähneln, ansonsten aber in der Regel nur einen sehr geringen Sprachumfang aufweisen. Dies gilt insbesondere für die Möglichkeit, komplexe Datenstrukturen definieren zu können, da ja möglichst auf vorhandene Komponenten und deren Funktionalitäten zurückgegriffen werden soll. Typisierte Variablen und Referenzen fehlen entweder ganz oder es ist bestenfalls eine dynamische Typprüfung vorgesehen. Diese Eigenschaften und die Tatsache, dass Skriptsprachen normalerweise interpretierte Sprachen sind, erlauben andererseits eine sehr flexible Programmierung, die insbesondere für das Rapid Prototyping gut geeignet ist. Insgesamt sind Skriptsprachen jedoch auf einem vergleichsweise niedrigen Abstraktionsniveau angesiedelt, so dass sie sich aus Sicht der Prozessmodellierung nur bedingt zur Spezifikation werkzeugübergreifender Abläufe eignen.

[12] früher OLE-Automation

3.3.3.3 Fazit

Als wesentliche Grundvoraussetzung, die in den Bereich der Kontrollintegration fällt, sollte ein Werkzeug alle seine über die Benutzeroberfläche zugreifbaren Funktionalitäten auch über eine programmierbare Schnittstelle offen legen [FeOh91; Poh*99]. Erst durch die Möglichkeit der programmgesteuerten Interaktion zwischen Werkzeugen lassen sich elementare Werkzeugdienste zu größeren Abläufen zusammenfügen.

Kommunikationsmechanismen wie Object Request Broker oder Message Server ermöglichen die Interaktion zwischen Werkzeugen auf der Basis programmierbarer Laufzeitschnittstellen. Die direkte Verwendung solcher Mechanismen führt jedoch dazu, dass Ablaufwissen als Geflechte von *Punkt-zu-Punkt-Beziehungen* im Programmcode der Werkzeuge versteckt bleiben. Die Interaktion zwischen den Werkzeugen sollte daher auf Basis expliziter Prozessmodelle von der Leitdomäne vermittelt werden.

In einigen prozesszentrierten Umgebungen wie SPADE oder ProcessWeaver „klinkt" sich die Prozessmaschine in den Nachrichtenverkehr zwischen den Werkzeugen ein und lässt die empfangenen Nachrichten in die Prozessmodellausführung einfließen. Problematisch ist hier, dass die verwendeten Werkzeugen unabhängig von ihrer Verwendung in einer prozesszentrierten Umgebung entwickelt wurden und eigene Reaktionen auf bestimmte Nachrichtentypen realisieren.

In komponentenbasierten Ansätze tritt die Grundidee der Entkopplung von Verarbeitung (Komponentenfunktionalität) und Koordination (Zusammenspiel der Komponenten) in Form von Koordinations- und Kompositionssprachen und spezieller Mediatordienste auf. Diese sind jedoch ebenso wie Architekturbeschreibungs- und Skriptsprachen nicht eigens für die Prozessmodellierung entwickelt worden und bewegen sich auf einem relativ niedrigen, technischen Abstraktionsniveau.

3.3.4 Beschreibung von Werkzeugdiensten

3.3.4.1 Motivation

Im vorangegangenen Abschnitt haben wir uns mit unterschiedlichen Mechanismen für die Werkzeuginteraktion zur *Laufzeit* beschäftigt und sind zum Schluss gekommen, dass Werkzeuginteraktion in der Durchführungsdomäne über die Leitdomäne, d.h. die Prozessmaschine, auf Basis explizit definierter Prozessfragmente vermittelt werden sollte. Diese Diskussion erfolgte aus der Perspektive des Werkzeugintegrators und war weitgehend unabhängig von der Frage, wie die Fähigkeiten eines Werkzeugs zur *Definitionszeit* repräsentiert und der Prozessmodellierung zugänglich gemacht werden.

Um sicherzustellen, dass die in einem Prozessfragment definierten Arbeitsschritte und Arbeitsmodi in der Durchführungsdomäne auch umgesetzt werden können, müssen die Werkzeuge bei der Prozessfragmentdefinition geeignet berücksichtigt werden. Dazu benötigt der Prozessmodellierer Informationen über die in einer Entwurfsumgebung vorhandenen Werkzeuge, ihre Dienste und deren Parameter, die Art des Dienstaufrufs etc. Diese Informationen bezieht der Prozessmodellierer typischerweise aus einer Vielzahl von Quellen, z.B. aus Handbü-

chern, Programmdokumentationen, formalen Schnittstellenbeschreibungen und Typbibliotheken oder auch schlicht aus persönlichem Wissen und individuellen Erfahrungen. Diese Vielfalt von Werkzeugbeschreibungen, die in heutigen heterogenen Arbeitsumgebungen eher die Regel als die Ausnahme ist, erschwert das Auffinden und die Zuordnung geeigneter Werkzeugunterstützung zu bestimmten Prozessschritten erheblich.

Überbrückung der konzeptuellen Distanz zwischen Prozess- und Werkzeugbeschreibungen

Eine wichtige Anforderung an die Prozessintegration von Werkzeugen besteht daher darin, Werkzeuge umfassend und auf der gleichen Abstraktionsebene wie Prozessfragmente zu repräsentieren, um so die *konzeptuelle Distanz* zwischen Prozess- und Werkzeugbeschreibungen zu verringern oder gar aufzuheben. Dadurch wird es möglich, einzelne Prozessschritte und Werkzeugdienste systematisch miteinander in Bezug zu setzen und Diskrepanzen, etwa das Fehlen einer benötigten Werkzeugfunktionalität oder die unzulässige Zuweisung eines Werkzeugdiensts zu einem Prozessschritts, aufzudecken. Weiterhin erlaubt erst eine integrierte Modellierung von Prozessen und Werkzeugdiensten ein konsistentes Änderungsmanagement. Beispielsweise lassen sich so beim Entfernen eines Werkzeugs bzw. beim Austausch durch ein anderes die Auswirkungen auf die Ausführbarkeit der existierenden Prozessfragmente analysieren.

Aus Sicht der Prozessmodellierung sind drei Aspekte eines Werkzeugs von besonderer Bedeutung:

- die *elementaren Dienste*, die ein Werkzeug über seine programmierbare Laufzeitschnittstelle zum Aufruf durch die Prozessmaschine für die Umsetzung eines Prozessschritts anbietet.
- die *Vorbedingungen*, denen die Ausführung eines elementaren Dienstes zu einem bestimmten Zeitpunkt unterliegt;
- die *Arbeitsmodi*, in die ein Werkzeug versetzt werden kann. Ein Arbeitsmodus entspricht dabei einer eingeschränkten Teilmenge von Diensten, unter denen der Benutzer zu einem Zeitpunkt auswählen kann. Wenn ein Werkzeug von der Prozessmaschine in einen Arbeitsmodus versetzt, hat dies *beratende* Wirkung, da der Benutzer nicht gezwungen wird, einen bestimmten Dienst auszuführen, sondern zwischen verschiedenen sinnvollen Prozessalternativen auswählen kann. Wir sprechen daher auch von *Beratungsdiensten*, die von einem Werkzeug angeboten werden.

Welche Aspekte sind werkzeuginhärent, welche sollten prozessadaptabel sein?

Hier stellt sich die Frage, welche Aspekte *werkzeuginhärent* sind, also durch ein gegebenes Werkzeug bereits festgelegt sind, und welche Aspekte von der Verwendung eines Werkzeugs innerhalb eines bestimmten Prozesses abhängen, also Teil der Prozessdefinition sein sollten. In [FeOh91] wird gefordert, dass ein Werkzeug einen prinzipiell *modusfreien* Zugriff auf seine Funktionalität erlauben sollte und per se möglichst wenige oder gar keine Annahmen darüber machen sollte, unter welchen Umständen seine Funktionalität exportiert werden kann. Einschränkungen hinsichtlich der Benutzbarkeit einzelner Dienste und die Definition spezieller, auf einen bestimmten Prozesskontext zugeschnittener Beratungsdienste sind prozessrelevante Aspekte und sollten aus Gründen der Anpassbarkeit daher logisch als Teil des Prozessmodell aufgefasst werden.

Die konzeptuelle Modellierung eines Werkzeuges kann sich daher zunächst auf die Angabe der *Signaturen* aller über die programmierbare Laufzeitschnittstelle aufrufbaren *elementaren Dienste* beschränken. Hierbei ist es zunächst gleichgültig, ob

die *Signaturen* im klassischen prozeduralen Stil dargestellt werden oder im objektorientierten Sinn als Objektmethoden bestimmter Werkzeugobjekte aufgefasst werden. Die Modellierung der Parameter berührt den Bereich der Datenintegration. Es muss sichergestellt werden, dass sich die Produkttypen der Dienstparameter auch im Produktmodell der Modellierungsdomäne berücksichtigt werden. Wie in Abschnitt 3.3.2 bereits erläutert wurde, gibt es hier zwei denkbare Vorgehensweisen. Entweder arbeiten Prozessmaschine und Werkzeuge auf den *gleichen* Daten in einem gemeinsamen Repository. Die übergebenen Parameter beziehen sich dann direkt auf Daten im gemeinsamen Repository. Im anderen Fall wird in Modellierungsdomäne auf der Schemaebene der prozessrelevante Teil der Werkzeugdatenmodelle nachmodelliert, wobei dann die Parameter lediglich Referenzen auf externe Daten der Werkzeuge darstellen, die nicht direkt im Repository der Modellierungsdomäne verwaltet werden.

Aufbauend auf einer konzeptuellen Beschreibung der Werkzeugdienste können dann im Prozessmodell elementare Dienste einzelnen Prozessschritten zugeordnet werden, Vorbedingungen für die Aktivierung von elementaren Diensten festgelegt werden und Beratungsdienste, d.h. Arbeitsmodi, definiert werden.

3.3.4.2 Bewertung existierender Ansätze

Bei der folgenden Literaturbetrachtung zur konzeptuellen Werkzeugbeschreibung sind drei Bereiche für uns von besonderer Relevanz. Wir diskutieren zunächst allgemeine Beschreibungstechniken für die Schnittstellen von Werkzeugen (bzw. Softwarekomponenten generell), insbesondere im Kontext heterogener, verteilter Systeme. Danach wenden wir uns der konzeptuellen Modellierung von Werkzeugen in existierenden Prozessformalismen zu. Insbesondere interessiert uns, ob und wie in existierenden prozesszentrierten Umgebungen extern vorliegende Schnittstellenbeschreibungen von Werkzeugen mit den Konzepten der zugrunde liegenden Prozessmodellierungssprachen integriert werden. Als dritten Teilaspekt betrachten wir abschließend Generierungsansätze, die mithilfe von Werkzeugspezifikationen, die auf vergleichsweise hohem logischen Niveau angesiedelt sind, die (semi-)automatische Erzeugung von Werkzeugfunktionalitäten erlauben.

Schnittstellenbeschreibungssprachen

Schnittstellenbeschreibungen von Softwarekomponenten treten auf Entwurfsebene in vielen Modellierungstechniken als zentrale Elemente auf, z.B. in Form von Klassenspezifikationen in objektorientierten Methoden wie OMT oder UML. Auf Implementierungsebene stellen Deklarationen von abstrakten Datentypen (in nicht objektorientierten Sprachen wie Pascal oder Modula) oder Klassen (in objektorientierten Sprachen wie C++, Java und Smalltalk) die elementarste Form von Schnittstellenbeschreibungen dar. Diese Schnittstellenbeschreibungen sind jedoch nur innerhalb einer homogenen programmiersprachlichen Umgebung anwendbar.

Interface Description Language

Zentraler Bestandteil von Infrastrukturen für verteilte (objektorientierte) Umgebungen wie z.B. CORBA oder DCOM sind daher so genannte *Interface Description Languages* (IDLs), mit deren Hilfe sich Schnittstellen von Softwarekomponenten *sprachneutral* notieren lassen. Auch die oben angesprochenen Architekturbeschreibungssprachen (siehe S. 91) bieten Sprachkonstrukte zu programmiersprachenunabhängigen Beschreibung von Komponentenschnittstellen an.

Die Beschreibung einer (Objekt-)Schnittstelle mit einer IDL dient zwei wesentlichen Zwecken. Zum einen werden solche Spezifikationen mithilfe von Generatoren, den so genannten IDL-Compilern, gemäß den in standardisierten Sprach-Bindings festgelegten Abbildungsregeln in Programmcode der Zielsprache transformiert. In CORBA existieren Sprach-Bindings für die gängigsten Sprachen wie z.B. C, C++, Java, Smalltalk und Ada. Dabei wird natürlich nicht die gesamte Objektfunktionalität erzeugt, sondern nur der Code für die Stubs und Skeletons (s.o.), der erforderlich ist für den Transport von Methodenaufrufen vom einem Rechner im Netz zu dem anderen, auf dem sich das aufgerufene Objekt befindet.

Generierung von Client-Stubs und Server-Skeletons

Zum anderen lassen sich IDL-Spezifikationen in zentrale Speicher für Schnittstellenbeschreibungen überführen, die u.a. Werkzeuge zur Verwaltung von Komponentensammlungen oder das Erfragen von Typinformationen zur Laufzeit unterstützen. In CORBA ist das *Interface-Repository* ein spezieller Service für die Verwaltung von Schnittstellenbeschreibungen, der über standardisierte CORBA-Methodenaufrufe den Laufzeitzugriff auf Schnittstellenobjekte erlaubt. Eine vergleichbare Aufgabe zur Beschreibung von Komponenten übernehmen in DCOM die *Type Libraries*. Interface-Repository (bzw. Type Library) und IDL-Datei sind lediglich zwei unterschiedliche Repräsentationsformen für die gleiche Information. Mithilfe geeigneter Werkzeuge können die Repräsentationen jeweils ineinander überführt werden. Ein etwas anderer Mechanismus liegt dem JavaBeans-Modell zugrunde, das Techniken zur Introspektion und Reflexion [KiRB91] nutzt und somit eher zur *Selbstauskunft* von Komponenten zur Laufzeit geeignet ist.

Interface-Repository zur Laufzeitunterstützung

In Message Broadcasting-Architekturen (siehe auch Abschnitt 3.3.3) sind Werkzeuge durch die Menge aller Nachrichtenmuster, für die sie sich beim Message Server registriert haben und auf die sie somit sinnvoll reagieren können, charakterisiert. Die Nutzbarkeit der Registrierungsinformationen zur Prozessmodellierung hängt stark davon ab, ob die Registrierung beim Message Server *statisch*, d.h. unabhängig von der Laufzeit eines Werkzeugs, oder *dynamisch* durch ein Werkzeug selbst, also z.B. beim Werkzeugstart, erfolgt. Im ersten Fall sind die Registrierungsinformationen außerhalb der Werkzeuge (in der Regel in Form von Konfigurationsdateien) zugänglich. Die dynamische Registrierung ist zwar flexibler, jedoch sind hier die Registrierungsinformation in den Werkzeugen versteckt (typischerweise in den Codeabschnitten für die Anmeldung beim Message Server) und somit zur Werkzeugbeschreibung bei der Definition von Prozessfragmenten wertlos. Ein Beispiel für eine rein statische Registrierung ist der Polylith Software Bus, während ToolTalk sowohl statische als auch dynamische Registrierung unterstützt. In Forest liegen die *policy descriptions* als erweiterte Werkzeugbeschreibungen explizit vor, können aber auch während der Laufzeit über das Policy Tool modifiziert werden. Eine Besonderheit stellen die bereits weiter oben erwähnten *Werkzeugprotokolle* in SoftBench dar, wo der Prozessmodellierer von einem Grundvorrat an *standardisierten* Werkzeugschnittstellen für bestimmte Werkzeugklassen ausgehen kann.

Werkzeugbeschreibungen in Message-Broadcasting-Architekturen

Werkzeugmodellierung in prozesszentrierten Umgebungen

In existierenden prozesszentrierten Umgebungen und Workflow-Management-Systemen ist die Konzeptwelt der zugrunde liegenden Prozessmodellierungssprachen durch Modellierungselemente wie *Aktivität*, *Produkt/Dokument*, *Rolle, Arbeitskontext* etc. geprägt (für eine Gegenüberstellung der wesentlichen Elemente verschiedener Prozessmodellierungssprachen siehe z.B. [Lonc94a; ABGM93;

McCh95; Lott93; FeHu93; CoJa99]). Eine explizite Repräsentation von Werkzeugen als *Objekten erster Klasse* findet dagegen im der Regel nicht statt.

Stattdessen spezifizieren viele prozesszentrierte Umgebungen den Aufruf externer Programme (Werkzeuge, Standardapplikationen) in der Leitdomäne über generische `execute`- oder `call`-Statements, deren Argumente die Namen von Werkzeugen und eventuell zusätzliche Startparameter beinhalten. Charakteristisch ist, dass diese Statements typischerweise als Attribute von Aktivitäten oder in deren Implementierungsspezifikation angegeben werden. Werkzeuge werden im Sinne der Blackbox-Integration (siehe Abschnitt 3.2) nur grobgranular und nicht auf der Ebene einzelner Dienste referenziert. Abb. 14 zeigt, angelehnt an [Böhm98], die Referenzierung von Werkzeugen in einem Prozessfragment. Die Darstellung bedient sich einer Pseudocode-Notation, die jedoch die typischen Merkmale der in PZEU und WFMS praktizierten Art der Werkzeugeinbindung gut illustriert:

***Abb. 14**: Werkzeugeinbindung in Prozessmodellierungssprachen (Pseudocode, vgl. [Böhm98])*

```
ProcessFragment SimpleProcessFragment
  SubStructures
    DataEntry: Activity EnterData;
    DataProcessing: Activity ProceedData;
  ControlFlow
    SEQUENCE(DataEntry, DataProcessing);
  ...
End SimpleProcessFragment

Activity EnterData
  EXEC "/usr/bin/formedit"
End EnterData

Activity ProceedData
  EXEC "/usr/bin/dataprocessing"
End ProceedData
```

Werkzeuge werden erst durch Referenzierung innerhalb des Prozessmodells „sichtbar"

Bei der Ausführung einer Aktivität (z.B. `EnterData`) wird der Pfad des aufzurufenden Werkzeugs (`"/usr/bin/formedit"`) als Argument der `EXEC`-Anweisung direkt an die Betriebssystemebene zur Aktivierung durchgereicht. Das Werkzeug ist also innerhalb des Prozessmodells nicht als konzeptuelles Objekt mit einer festgelegten Schnittstelle, sondern lediglich durch einen uninterpretierten String repräsentiert. Dadurch ist z.B. unmöglich, unzulässige Verwendungen des Werkzeugs, etwa aufgrund nicht passender Parameterlisten, zu entdecken. Außerdem existiert keine *von konkreten Prozessdefinitionen unabhängige* Modellierung der verfügbaren Werkzeuge bzw. Werkzeugdienste, d.h. ein Werkzeug wird innerhalb des Prozessmodell erst „sichtbar", wenn es innerhalb einer `Call/Exec`-Anweisung referenziert wird. Der Prozessmodellierer erhält somit keine Unterstützung bei der Suche nach passender Werkzeugfunktionalität für einen Prozessschritt, was insbesondere in großen Entwurfsumgebungen mit einer Vielzahl von Werkzeugen die konsistente Zuordnung von Prozessschritten und Werkzeugen erheblich erschwert.

Vor- und Nachbereitung des Werkzeugaufrufs durch Wrapper

In vielen Fällen müssen spezielle Vor- und Nachbereitungsmaßnahmen für einen externen Werkzeugaufruf getroffen werden, so dass ein direktes Durchreichen des `Call/Exec`-Statements an die Betriebssystemebene nicht ausreicht. Aus diesem Grund sehen viele Ansätze die Kapselung von Werkzeugen durch „Werkzeugumschläge", so genannte *Envelopes* oder *Wrapper*, vor [JaBu96; Böhm98]. Anstatt das

Werkzeug direkt zu referenzieren, wird im `Call`-Statement der entsprechende Wrapper angegeben und bei der Ausführung der zugehörigen Aktivität gestartet. Beispielsweise muss ein Wrapper in den Fällen, in denen die Prozessmaschine und das Werkzeug nicht auf einem gemeinsamen Repository für die Produktdaten arbeiten – bei der Blackbox-Integration ist dies fast immer der Fall –, die im Repository abgelegten Produktdaten vor dem Werkzeugstart zunächst in einen Arbeitsbereich transferieren, auf den das Werkzeug Zugriff hat. Meist geschieht dies in der Form, dass die betreffenden Dokumente aus dem Repository kopiert und in einem vorher spezifizierten Pfad des Dateisystem abgelegt werden. Analog ist der Wrapper dafür verantwortlich, modifizierte oder neu erzeugte Dokumente nach der Werkzeugausführung in das Repository zurückzuspielen. Außer zur Datenversorgung des aufgerufenen Werkzeugs werden Wrapper manchmal auch zu weiteren vorbereitenden Maßnahmen (Terminal-Emulation, Login-Skripte, Auswahl eines Netzknotens unter dem Gesichtspunkt der Lastbalancierung etc.) verwendet [Böhm98].

Für die Realisierung von Wrappern werden in der Regel Shell-Skripte verwendet. Problematisch ist dabei, dass wesentliche Werkzeugcharakteristika, zum Beispiel der Bezug zwischen den Werkzeugparametern und den Produktdaten im Prozess-Repository, *ad hoc* und *innerhalb* des Shell-Skripts behandelt werden und auf der Prozessmodellierungsebene nicht sichtbar sind, da Shell-Sprachen keine explizite Deklaration einer Ein-/Ausgabeschnittstelle vorsehen und lediglich die Rückgabe eines einzelnen Integer-wertigen Statuscodes erlauben, der nur Aufschluss über die erfolgreiche oder fehlgeschlagene Ausführung des Werkzeugs gibt. Die so spezifizierten Referenzen auf externe Werkzeuge haben aus Sicht der Prozessmodellierung keine Semantik.

Zum Teil werden diese Defizite durch die erweiterte Shell-Sprache *SEL* (Shell Envelope Language) [GiKa91] behoben, welche für die auf einem Regelformalismus basierende prozesszentrierte Umgebungen Marvel [BaKa91] und Oz [BeKa98] speziell zum Blackbox-Wrapping von Werkzeugen entwickelt wurde. Ein in *SEL* spezifizierter `Envelope` hat folgenden Aufbau (siehe Abb. 15):

```
ENVELOPE name
SHELL ksh | csh | sh;
INPUT
        type_x1 : X_1;
        ...
        type_xm : X_m;
OUTPUT
        type_y1 : Y_1;
        ...
        type_Yn : Y_n;
BEGIN
        <Body specified in shell language>
        RETURN status_code : Y1,…, Yn;
END
```

Abb. 15: *Gerüst einer SEL-Wrapperspezifikation [GiKa91]*

Aus Sicht der Modellierungsdomäne wird die Durchführungsdomäne als eine Menge von `Envelopes` repräsentiert, die durch die Angabe ihrer `Input`- und `Output`-Parameter charakterisiert sind. Die Typen der Parameter beziehen sich dabei auf das Datenmodell der Modellierungsdomäne, also die Objekttypen der in Mar-

vel definierten Objektbank. Dadurch sind statische Typüberprüfungen von Envelope-Schnittstellen und Regelspezifikationen möglich. Der eigentliche Wrappercode, der den Aufruf des Werkzeugs sowie die eventuell erforderlichen vor- und nachbereitenden Maßnahmen enthält, wird innerhalb des `Begin-End`-Blocks in einer üblichen Shell-Sprache spezifiziert und bei der Ausführung an den in der `Shell`-Klausel angegebenen Kommandointerpreter (`ksh`, `csh` oder `sh`) übergeben. Als nachteilig erweist sich hier, dass in SEL zwar die Envelopes, nicht aber die Werkzeuge selbst explizit modelliert werden. Dies macht es beispielsweise schwierig, beim Entfernen eines Werkzeugs aus einer Umgebung die Auswirkungen auf die Prozessdefinitionen zu bestimmen.

Dieser Nachteil wird durch eine Erweiterung von SEL, dem *Multi Tool Protocol* (MTP) [VaKa96], behoben. MTP führt als weitere Beschreibungsebene die explizite Darstellung von Werkzeugen ein (siehe Abb. 16). Einem Werkzeug zugeordnet werden alle Aktivitäten, in denen das Werkzeug verwendet wird. Für jede Aktivität ist hier die Angabe eines spezifischen Envelopes mit jeweils eigenen Parameterlisten möglich.

***Abb. 16**: Werkzeugmodellierung in MTP [VaKa96]*

```
<tool-name> :: superclass TOOL;
  [ path          : string;
    architecture  : (sun4, solaris, …);
    host          : string;
    instances     : integer;
    multi-flag    : (Uni/Queue, Multi/Queue,
                     Uni/NoQueue, Multi/NoQueue);
  ]
  <activity-name> : string =
       "<envelope-name> <parameters locks>";
  <activity-name> : string =
       "<envelope-name> <parameters locks>";
  …
end
```

Darüber hinaus wird der Aufruf und die Interaktion mit einem Werkzeug durch eine Reihe von zusätzlichen Attributen genauer charakterisiert. Die Attribute `architecture`, `host` und `path` geben an, unter welchen Betriebsystemen das Werkzeug (bzw. der Envelope) läuft, auf welchem Netzwerkknoten es gestartet werden soll und unter welchem Netzwerkpfad das Executable zu finden ist. Mit dem Attribut `instances` lässt sich die maximale Anzahl gleichzeitig laufender Werkzeuginstanzen auf einen bestimmten Wert begrenzen, etwa aufgrund einer beschränkten Anzahl von Lizenzen oder um bei ressourcenintensiven Werkzeug eine Überlast zu verhindern. Am interessantesten ist das `multi-flag`-Attribut, das das Kreuzprodukt aus zwei orthogonalen Dimensionen annehmen kann (`Uni/Queue`, `Uni/NoQueue`, `Multi/Queue`, `Multi/NoQueue`). `Uni` zeigt an, dass eine Werkzeuginstanz nur von einem Benutzer verwendet werden kann, während `Multi` bedeutet, dass das Werkzeug über Multi-User-Fähigkeiten verfügt und von mehreren Benutzern gleichzeitig bedient werden kann. `Queue` legt fest, dass zu einem Zeitpunkt nur eine Aufgabe in einem Werkzeug durchführt werden kann, während bei `NoQueue` ein Werkzeug simultan in mehreren unterschiedlichen Aufgaben verwendet werden kann. Die `Queue`- bzw. `NoQueue`-Eigenschaft wird im Wesentlichen zurückgeführt auf die Fähigkeit eines Werkzeugs, lediglich einen oder mehrere Dokument-Buffer gleichzeitig verwalten zu können (analog zur Unterscheidung zwischen Single Document Interface-Applikationen (SDI) und Multiple Document

Interface-Applikationen (MDI) im Windows MFC-Rahmenwerk [13]). Wichtig ist, dass eine direkte Interaktion zwischen dem Wrapper und dem Werkzeug, sobald das Werkzeug erst einmal gestartet wurde, nicht vorgesehen ist. Stattdessen wird der Benutzer über ein separates Dialogfenster über Aktionen informiert, die er in einem laufenden Werkzeug dann manuell vornehmen muss. Wenn beispielsweise gemäß der Prozessmodellausführung die in einem Werkzeug aktuell bearbeitete Aufgabe gewechselt werden soll (beim NoQueue-Modus), wird der Benutzer über ein Informationsfenster angewiesen, im betreffenden Werkzeug den aktuellen Dokument-Buffer zu tauschen. Der für ein Werkzeug bzw. für eine Aufgabe spezifische Inhalt des Informationsfensters wird innerhalb einer MTP-Spezifikation definiert.

WfMC-Standards

Ansätze wie SEL/MTP legen somit zwar die Schnittstellen von Werkzeug-Wrappern für die Prozessmodellierung offen, unterstützen aber lediglich die Blackbox-Integration von Werkzeugen. Auch die Workflow Management Coalition (WfMC), ein herstellerübergreifendes Gremium, das Standards für die Interoperabilität von Workflow-Managementsystemen vorantreibt, beschränkt sich auf die Blackbox-Integration von Werkzeugen. Die WfMC hat in der *Workflow Management API (Interface 2&3) Specification* [WFMC98a] Schnittstellen für den Informationsaustausch zwischen Komponenten eines WFMS (insbesondere Prozessmaschine und Agenda-Werkzeuge) einerseits und externen Applikationen andererseits festgelegt. Ähnlich der Integrationsstrategie des oben beschriebenen SEL/MTP-Ansatzes geht die WfMC davon aus, dass die Prozessmaschine nicht direkt auf externe Applikationen zugreift, sondern über Wrapper, die in der WfMC-Terminologie als *Tool Agents* bezeichnet werden. Die Hauptaufgabe der Tool Agents besteht in der Abstraktion von unterschiedlichen Aufrufmechanismen (Kommandozeile, RPC, CORBA etc.) für die einzubindenden Applikationen (siehe Abb. 17).

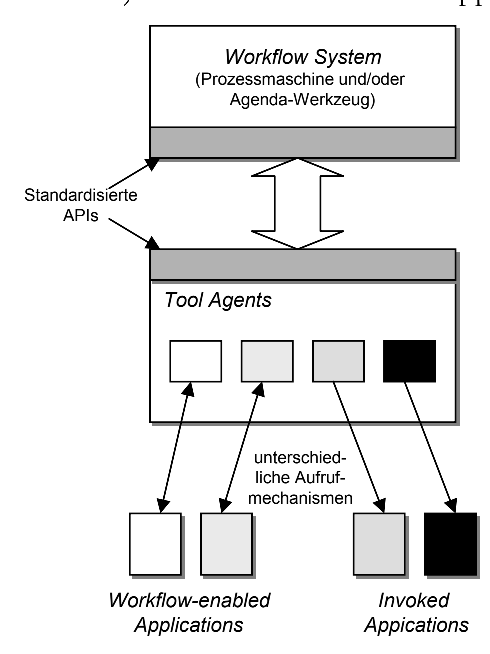


***Abb. 17**: WFMC-Schnittstellen zur Applikationsintegration [WFMC98a]*

Die WfMC unterscheidet zwischen normalen Applikationen, die nicht eigens für den Einsatz innerhalb eines WFMS konzipiert wurden, und so genannten *Workflow-enabled Applications*, die von Workflow-spezifischen Funktionalitäten

[13] MFC: Microsoft Foundation Classes

Gebrauch machen können. Insbesondere für die letztere Klasse von Applikationen sieht die WAPI (Interface 2&3) Specification eine Schnittstelle *zum* Workflow-System vor, über die beispielsweise Informationen über die aktuell laufenden Aktivitätsinstanzen erfragt werden können. Während diese Schnittstelle recht reichhaltige Funktionen anbietet, ist die Schnittstellenspezifikation für die Tool Agents nur äußerst rudimentär ausgearbeitet und auf die grobgranulare Blackbox-Integration von Applikationen ausgerichtet. Die dort spezifizierten Funktionen betreffen lediglich das Starten und Beenden einer angegebenen Applikation und das Erfragen aktueller Statusinformationen des Tool Agents. Eine inkrementelle Aktivierung individueller Dienste der aufgerufenen Applikation zur Laufzeit ist nicht vorgesehen.

Die bislang betrachteten Ansätze unterstützen nur die grobgranulare Blackbox-Integration und repräsentieren Werkzeuge – wenn überhaupt – lediglich als monolithische Operatoren, die gestartet und ggf. beendet werden können. Wie bereits in Abschnitt 3.3.3 angesprochen wurde, macht eine zunehmende Zahl prozesszentrierter Umgebungen Gebrauch von Kontrollintegrationsinfrastrukturen wie Object Request Broker oder Message Broadcasting-Mechanismen.

Unintegrierte Koexistenz von Werkzeugbeschreibungen in Interface Repositories und Prozessmodellen

Hier läge es nahe, die in den Interface Repositories bzw. Message Servern vorliegenden Werkzeugbeschreibungen systematisch bei der Modellierung von Prozessschritten zu berücksichtigen. Dies ist jedoch nicht der Fall, wie das bereits weiter oben erwähnte Beispiel der SPADE-Umgebung mit der Prozessmodellierungssprache SLANG zeigt. Als Sprachmittel zur Beschreibung der Interaktion mit der Durchführungsdomäne stehen in SLANG die Konzepte *black transition* und *user interface place* zur Verfügung. Das Schalten einer *black transition* führt zur Aktivierung eines externen Programms, welches im Implementierungsteil einer schwarzen Transition spezifiziert wird. Es ist nicht möglich, einen Werkzeugdienst bzw. eine beim Message Server registrierte Nachricht direkt in der Spezifikation der schwarzen Transition zu referenzieren. Stattdessen ruft die *black transition* ein Wrapper-Programm (`SendMessage`) auf, welches seinerseits eine Nachricht an den Message Server versendet und so eine Reaktion in der Werkzeugumgebung auslöst. Analog verläuft der Empfang von Nachrichten aus der Durchführungsdomäne, die in *user interface places* der interpretierten SLANG-Netze abgebildet werden. In ProcessWeaver erfolgt die Interaktion mit nachrichtenbasierten Werkzeugen auf ähnliche Weise über die Einbettung von `CoShell`-Prozeduren in Prozessfragmente. `CoShell`-Prozeduren werden in einer an UNIX-Shellsprachen angelehnten Sprache spezifiziert, welche über zusätzliche Bibliotheksfunktionen zum Empfang und Absenden von Nachrichten von bzw. an einen Message Server verfügt.

Mithilfe solcher Mechanismen können Werkzeuge im Vergleich zur reinen Blackbox-Integration zwar auf der Ebene individueller Werkzeugdienste integriert werden. Bei der Modellierung von Prozessfragmenten steht dem Prozessmodellierer jedoch kein konzeptuelles Modell der vorhandenen Werkzeugdienste zur Verfügung. Vielmehr koexistieren Prozessbeschreibung in der Modellierungsdomäne (z.B. SLANG-Netze) und Werkzeugbeschreibungen der zugrunde liegenden Kontrollintegrationsinfrastrukturen (z.B. Message Server-Repositories) *unintegriert* nebeneinander. Das einzige Bindeglied stellen *ausprogrammierte* Wrapper (z.B. `SendMessage` bei SPADE/SLANG oder `CoShell`-Skripte in ProcessWeaver) dar. Genau wie bei Blackbox-Ansätzen geschieht die modellierungsseitige Einbindung von Werkzeugen also *ad hoc*.

Werkzeuggenerierung

In den bislang diskutierten Ansätzen dienen Schnittstellenbeschreibungen von Werkzeugdiensten lediglich als Grundlage für die Referenzierung von Dienstaufrufen in Prozessmodellen und deren Bindung an Dienstimplementierungen, die von Werkzeugentwicklern bereitgestellt wurden. Einen Schritt weiter gehen Ansätze zur Werkzeuggenerierung, die die (semi-)automatische Erzeugung von Werkzeugen auf Basis einer formalen, semantisch angereicherten Spezifikation der zugrunde liegenden Datenstrukturen und Dienste zum Ziel haben.

Frühe Beispiele für Werkzeuggenerierungsansätze sind syntaxgesteuerte Editoren, z.B. der *Cornell Program Synthesizer* [ReTe81; ReTe88] und die Werkzeuge der Umgebungen *Gandalf* [HaNo86] und *Mentor* [DHKL84]. Speziell auf den Einsatz innerhalb einer prozesszentrierten Umgebung ausgerichtet ist die im Rahmen des GOODSTEP-Projekts [GOOD94] entwickelte Werkzeugspezifikationssprache GTSL[14] [Emme95; Emme96; EAMP97]. GTSL folgt einem datenbankzentrierten Entwurfsparadigma und legt den Schwerpunkt auf die Generierung von Datenbankschemata, Konsistenzchecks und darauf basierenden elementaren Werkzeugdiensten. Die formale Grundlage stellen projektweite, abstrakte Syntaxgraphen für die zu unterstützenden Entwurfsdokumente dar, die auf ein objektorientiertes Datenmodell abgebildet werden. Die Laufzeitumgebung eines GTSL-basierten Werkzeugs bietet eine Kommunikationsschnittstelle, über die eine externe Prozessmaschine die generierten Werkzeugdienste und Konsistenzchecks aufrufen kann. GTSL liefert somit Lösungen für die Datenintegration und den feingranulare Aufruf von Werkzeugdiensten. Die prozesssensitive Anpassung des Dienstangebots eines Werkzeugs gemäß der Prozessmodellausführung durch die Prozessmaschine (siehe Abschnitt 3.3.6) sowie die Aktivierung von Prozessfragmenten aus den Werkzeugen (siehe Abschnitt 3.3.7) werden jedoch nicht direkt unterstützt. Aus Benutzersicht bieten GTSL-basierte Werkzeuge zwar kontextsensitive Menüs an. Deren Inhalt ist jedoch ausschließlich durch den aktuellen Zustand des zugrunde liegenden abstrakten Syntaxgraphen, das aktuell selektierte Produktinkrement und die in der Werkzeugspezifikation angegebenen Aktionsregeln bestimmt.

GTSL

Weitere Beispiele für Werkzeuggeneratoren kommen aus der Forschung über Method Engineering und MetaCASE-Umgebungen [Alde91; KeSm96; Brin96]. Das primäre Ziel dieser Ansätze liegt in der schnellen Anpassbarkeit von Werkzeugumgebungen an spezifische Modellierungs*notationen*. Grundlage der meisten MetaCASE-Umgebungen ist die Modellierung der zu unterstützenden Notationen als Instanz eines generischen Metamodells, aus dem eine Menge von spezifischen Werkzeugdiensten zur Modellmanipulation abgeleitet werden. Das Dienstangebot eines so generierten Werkzeugs hängt jedoch nur von der Struktur der jeweiligen Modellierungsnotation ab und ist nicht an explizite Prozessdefinitionen gekoppelt. Zudem ist das Spektrum der von einer MetaCASE-Umgebung unterstützten Notationen stark vom zugrunde liegenden Metamodell beschränkt. So lassen sich beispielsweise innerhalb der MetaEdit-Umgebung mithilfe des GOPRR-Metamodells [KeLR95] lediglich so genannte „bubble-and-arc“-Notationen sehr gut beschreiben, während hierarchisch strukturierte Techniken (z.B. geschachtelte Pa-

MetaCASE-Umgebungen

[14] GTSL: GOODSTEP Tool Specification Language

ketdiagramme) und Diagramme, in denen das Layout semantische Information trägt (z.B. Message Sequence Charts), nur schlecht unterstützt werden.

Zusammenfassend beruhen existierende Generierungstechniken stark auf einer strikten Formalisierung der einer Methodik zugrunde liegenden Produktstruktur. Das generelle Problem generativer Ansätze besteht also darin, dass Generatoren erst dann erfolgreich gebaut werden können, wenn der adressierte Problembereich sehr gut verstanden ist. Darüber hinaus sind Produkte unterschiedlicher Generatoren häufig schwer zu integrieren, da sie stark von den jeweils zugrunde liegenden Automatismen und Grundgerüsten geprägt sind [Grif98]. Beispielsweise macht der GTSL-Ansatz weitreichende Annahmen über die zugrunde liegenden Datenintegrationstechnik, nämlich die Integration über projektweite abstrakte Syntaxgraphen, und beschränkt sich auf jeweils syntaxgerichtete und textuelle Werkzeuge.

3.3.4.3 Fazit

Um die Unterstützungsleistung der in einer Umgebung vorhandenen Werkzeuge angemessen bei der Modellierung von Prozessfragmenten berücksichtigen zu können, muss eine konzeptuelle Repräsentation der Werkzeuge vorliegen. Als Minimalanforderungen sollten Werkzeuge durch Angabe der Signaturen ihrer von außen aufrufbaren Dienste dargestellt werden.

In existierenden Prozessmodellierungssprachen werden Werkzeuge meist als monolithische Operatoren betrachtet und nicht als Objekte erster Klasse repräsentiert. Werkzeugen werden in Prozessmodellen *ad hoc* berücksichtigt, d.h. sie werden auf Prozessmodellierungsebene erst dann sichtbar, sobald sie zum ersten Mal innerhalb des Prozessmodells zur Umsetzung einer Aktivität referenziert werden. Zudem werden die Werkzeuge häufig innerhalb von Wrappern gekapselt. In Ansätzen, die Werkzeuge feingranular auf der Basis von Kontrollintegrationsinfrastrukturen wie Object Request Broker und Messsage Server einbinden, koexistieren Prozess- und Werkzeugdienstbeschreibungen in voneinander unabhängigen Prozess- bzw. Schnittstellenrepositories ohne eine ausreichende Unterstützung der Konsistenzsicherung beider Modelle.

Generative Ansätze zur Werkzeugerstellung gehen über die reine Werkzeugbeschreibung hinaus und erfordern eine strikte Formalisierung der zugrunde liegenden Produktstrukturen. Ihr Einsatzgebiet beschränkt daher auf sehr gut verstandene Modellierungstechniken mit einem großen Automatisierungspotenzial, in denen die in dieser Arbeit angestrebte Prozessadaptibilität für kreative Entwurfsprozesse ohnehin an Bedeutung verliert.

3.3.5 Synchronisation zwischen den Prozessdomänen

3.3.5.1 Motivation

Drei Prozesssichten

Die Trennung zwischen Modellierungs-, Leit- und Durchführungsdomäne in Dowson's Domänenmodell für prozesszentrierte Umgebungen ([DoFe94], siehe Abschnitt 2.2.4.2) hebt den Unterschied zwischen drei fundamentalen Sichten auf den Prozess hervor [Cugo98]:

- **Prozessmodell** (Modellierungsdomäne): Dies ist ein in einem bestimmten Formalismus notiertes Modell des *erwarteten* Prozesses und liefert eine *statische* „Soll"-Sicht auf den Prozess.
- **Realer Prozess** (Durchführungsdomäne): Dies ist der eigentliche Prozess, so wie er in der realen Welt durchgeführt wird. Der reale Prozess lässt sich charakterisieren durch den aktuellen Zustand der Produkte, auf denen der Prozess operiert, sowie durch die Prozesshistorie, also die Abfolge von Ereignissen und Aktivitäten, seit der Prozess gestartet wurde. Der reale Prozess ist eine *dynamische* Entität, d.h. er ändert sich, sobald ein neues Ereignis eintritt (z.B. Beendigung einer Aktion).
- **Beobachteter Prozess** (Leitdomäne): Während der Prozessmodellausführung hat die Leitdomäne nur eine partielle, vergröberte Sicht auf den realen Prozess. Der beobachtete Prozess basiert auf Aktionen, die der Benutzer unter Kontrolle der Leitdomäne durchführt bzw. über die die Leitdomäne explizit informiert wird. Alle anderen Ereignisse, die den realen Prozess beeinflussen, sind für die Leitdomäne nicht sichtbar. Zu jedem Zeitpunkt kann der beobachtete Prozess durch die Historie von Aktivitäten beschrieben werden, die der Benutzer unter Kontrolle der Leitdomäne durchgeführt hat. Ebenso wie der reale Prozess stellt der beobachtete Prozess eine *dynamische* Sicht dar.

In einer prozessintegrierten Umgebung müssen die Sichten in den verschiedenen Prozessdomänen *synchronisiert* werden und mögliche Abweichungen und Inkonsistenzen vermieden oder geeignet behandelt werden. Konflikte zwischen den Prozesssichten können in zwei fundamental verschiedene Kategorien eingeteilt werden: Modellinkonsistenzen und Beobachtungsinkonsistenzen[15].

Modellinkonsistenzen

Eine Modellinkonsistenz tritt auf, wenn die reale Prozessdurchführung vom erwarteten, d.h. laut Prozessmodell „erlaubten" Verhalten, abweicht. Dies ist zum Beispiel der Fall, wenn ein Entwickler mit der Implementierung eines Quellcode-Modules beginnt, ohne vorher einen Entwurf für dieses Modul erstellt zu haben, obwohl dies im Prozessmodell so vorgesehen war. Eine Modellabweichungen verletzt also im Prozessmodell spezifizierte Randbedingungen und Einschränkungen.

Für das Auftreten von Modellabweichungen gibt es zwei qualitativ unterschiedliche Gründe. Eine mögliche Ursache ist, dass ein Entwickler seiner eigenen Urteilskraft hinsichtlich des weiteren Vorgehens mehr vertraut als der Präskription durch das Prozessmodell oder dass eine Situation auftritt, die zum Zeitpunkt der Erstellung des Prozessmodell noch nicht in Betracht gezogen wurde (z.B. wenn aufgrund terminlicher oder personeller Engpässe Prioritäten neu gesetzt und eigentlich sinnvolle Schritte ausgelassen werden müssen). In diesen Fällen weicht der Entwickler *bewusst* von den Vorgaben des Prozessmodells ab. Eine prozessintegrierte Umgebung sollte *bewusste* Modellabweichungen des Benutzers vom vorgegebenen Prozess *grundsätzlich* zulassen und unterstützen. Nachdem frühe Prozess-

Bewusste Prozessabweichungen sollten zugelassen und entsprechend behandelt werden

[15] In [CDFG96; Cugo98] werden hierfür die Begriffe *domain-level-* bzw. *environment level* inconsistencies verwendet.

unterstützungsansätze anfangs stark auf Automatisierung und strikte Prozesslenkung ausgerichtet waren, hat sich mittlerweile die Ansicht durchgesetzt, dass letztendlich der menschliche Benutzer das Prozessgeschehen diktieren sollte und nicht das Prozessmodell [Redw93; WALM99; Pohl97]. Dies trifft gerade in Entwurfsdomänen mit einer Vielzahl kreativer Entscheidungen zu, wo strikte Unterstützungsansätze schnell als zu unflexibel und rigide abgelehnt werden. Hier können Prozessabweichungen vielmehr eine wichtige Informationsquelle für die Prozessverbesserung darstellen [JPRS94; Hump89]. Es hängt allerdings auch vom konkreten Anwendungskontext ab, in welchem Maße Abweichungen vom intendierten Prozess tolerierbar sind. Bei gutverstandenen und repetitiven Abläufen, die vornehmlich in betrieblichen Prozessen im Rahmen von Workflowsystemen unterstützt werden, ist ein hoher Durchsetzungsgrad der Prozessunterstützung eher angemessen. Das gleiche kann aber auch in kreativen Prozessen gelten, wenn die Konformität der Prozessdurchführung zu definierten Prozessmodellen zwischen Auftragnehmern und Kunden vertraglich vereinbart wurde. Dies ist zum Beispiel dann der Fall, wenn zur Gewährleistung von Nachvollziehbarkeit Entwicklungsaktivitäten prozessbegleitend dokumentiert werden müssen [Pohl99; Dömg99].

Unbewusste Verletzungen der Prozessmodellvorgaben

Eine andere Ursache für das Entstehen von Modellinkonsistenzen kann in einer mangelhaften Repräsentation der aktuell gültigen Prozessvorgaben in der Durchführungsdomäne liegen, d.h. es ist für den Entwickler in seiner Arbeitsumgebung nicht oder nur schwer erkennbar, welche Prozessschritte im aktuellen Prozesszustand prozessmodellkonform sind und welche Schritte im Widerspruch zu den Einschränkungen des Prozessmodells stehen. In diesem Fall würde der Entwickler die Vorgaben aus dem Prozessmodell *unbewusst* verletzen. Im Gegensatz zu beabsichtigten Abweichungen sollten unbewusste Modellabweichungen unbedingt vermieden werden, da ansonsten die gesamte Prozessunterstützung wertlos wird. Voraussetzung dafür ist, dass die Werkzeugumgebung die aktuell erlaubten Prozessschritte adäquat in der Benutzeroberfläche reflektiert und den Zugriff auf nicht vorgesehene Schritte nur auf explizite und bewusste Anforderung des Benutzers erlaubt. Die daraus resultierende Forderung nach prozesssensitiven Benutzerschnittstellen behandeln wir in Abschnitt 3.3.6.

Beobachtungsinkonsistenzen

Divergenz zwischen realem und beobachtetem Prozess

Eine Beobachtungsinkonsistenz ist dann gegeben, wenn der beobachtete Prozess keine korrekte Sicht auf den realen Prozess wiedergibt. Solche Inkonsistenzen sind darauf zurückzuführen, dass prozessrelevante Ereignisse in der Durchführungsdomäne nicht oder nur unkorrekt an die Leitdomäne bekannt gegeben oder dort berücksichtigt werden. Beispielsweise könnte ein Entwickler den Entwurf für ein Modul fertiggestellt haben, ohne dass die Leitdomäne davon Kenntnis erlangt. Während der Entwickler gemäß dem Prozessmodell aus der Modellierungsdomäne nun eigentlich mit der Implementierung beginnen könnte, wartet die Leitdomäne noch auf die in Wirklichkeit bereits erfolgte Fertigstellung des Modulentwurfs. Abweichungen dieser Art werden als Beobachtungsabweichungen bezeichnet. Im Gegensatz zu bewussten Modellabweichungen, die letztendlich auf einen inhaltlichen Konflikt zwischen Prozessmodell und realem Prozess hindeuten, sollten Beobachtungsabweichungen auf jeden Fall vermieden werden, da von der Leitdomäne auf der Basis eines unzutreffenden internen Abbilds des realen Prozesszu-

stands kaum sinnvolle Prozessunterstützung und -kontrolle erwartet werden kann [DoFe94; Fern93a; Mont94; CDFG96; Cugo98; BaKr95] .

Feedback und Synchronisationsprotokolle

Um Abweichungen zwischen dem realen und dem beobachteten Prozess zu vermeiden, muss das Zusammenspiel zwischen Leit- und Durchführungsdomäne durch ein definiertes und von beiden Seiten respektiertes *Synchronisationsprotokoll* koordiniert werden. Ein solches Protokoll muss nicht nur Aufrufbeziehungen von der Leitdomäne zur Durchführungsdomäne im Sinne einer klassischen Client-Server-Interaktion vorsehen (Aufruf einer Werkzeugaktion und Auswertung der Ergebnisse), sondern muss dafür Sorge tragen, dass auch sonstige prozessrelevante Ereignisse und Informationen aus der Durchführungsdomäne laufend an die Leitdomäne zurückgeliefert werden. Am natürlichsten lassen sich die prozessrelevanten Ereignisse direkt aus den Werkzeugen in den Durchführungsdomäne ableiten, da der Zustand der Werkzeuge (geladene Dokumente) und die dort stattfindenden Ereignisse (Selektion von Objekten, Aktivierung von Kommandos) den aktuellen Prozess reflektieren.

Synchronisationsprotokoll zum Abgleich zwischen realem und beobachtetem Prozess

Die erforderlichen Rückmeldungen hängen vom aktuellen Zustand der Prozessmodellausführung ab. Da wir davon ausgehen, dass die Prozessunterstützung nicht durchgängig, sondern nur für fragmentarisch modellierte Prozessabschnitte aktiv wird, müssen wir zwischen einem *reaktiven* und einem *proaktiven* Modus der Prozessunterstützung unterscheiden [BaDF96].

Im reaktiven Modus kann der Benutzer frei mit seinen Werkzeugen interagieren, bis ein bestimmter Zustand erreicht wird, in dem ein definiertes Prozessfragment zur Unterstützung zur Verfügung steht und von den Werkzeugen der Durchführungsdomäne angefordert wird (siehe auch Anforderungen „Werkzeugunterstützter Aufruf von Prozessfragmenten“ in Abschnitt 3.3.7).

Im proaktiven Modus kontrolliert die Leitdomäne das Geschehen in der Durchführungsdomäne, indem sie Dienste in den Werkzeugen aktiviert oder diese in Arbeitsmodi mit eingeschränktem Funktionsumfang versetzt. Nach dem Aufruf eines Werkzeugdienstes erwartet die Leitdomäne generelle Informationen über die erfolgreiche bzw. fehlgeschlagene Ausführung des Dienstes sowie über die Rückgabewerte des Dienstes. Im Falle der Aktivierung eines Arbeitsmodus in einem Werkzeug muss die Leitdomäne dagegen über die getroffene Benutzerauswahl, d.h. welcher Dienst auf welchen Produkten ausgewählt wurde, informiert werden.

Weiterhin muss die Leitdomäne über *bewusste* Abweichungen von den Vorgaben des Prozessmodells informiert werden, um die aktuelle Prozessfragmentausführung abbrechen oder eine Ausnahmebehandlung einleiten zu können.

3.3.5.2 Bewertung existierender Ansätze

In existierenden prozesszentrierten Umgebungen und Workflowmanagementsystemen sind im Wesentlichen zwei Ansätze bekannt, um die Leitdomäne mit Informationen aus der Durchführungsdomäne zu versorgen: Explizite Rückmeldungen über eine spezifische Benutzerschnittstelle oder Abhorchen von Ereignissen in der Durchführungsdomäne.

In den meisten Umgebungen erfolgt die Interaktion mit der Leitdomäne über spezifische Assistenzwerkzeuge wie Task-Manager, Agendawerkzeuge, Arbeitskontexte etc. (siehe auch Abschnitt 2.2.4.3). Über diese Werkzeuge nimmt der Benutzer Informationen über anwendbare Schritte entgegen und liefert Rückmeldungen über beendete Prozessschritte. Problematisch ist hierbei zum einen, dass der Benutzer neben seinen eigentlichen Werkzeugen mit einer zusätzlichen Benutzerschnittstelle konfrontiert wird. Je feiner die Arbeitsschritte und Produkte sind, auf die sich die Prozessunterstützung bezieht, desto höher wird der Aufwand für den Benutzer für die Interaktion mit den Assistenzwerkzeugen. Insbesondere die Eingabe aussagekräftiger Feedback-Informationen erfordert eine Referenzierung von Produktteilen unterhalb der Dokumentebene und ist bei separaten Benutzerschnittstellen sehr schwierig und mühselig. Da der Benutzer außerdem selbst für die Rückmeldungen an die Leitdomäne zuständig ist, besteht die Gefahr der oben genannten Modellabweichungen aufgrund falscher, idealisierter oder nicht zeitnah gelieferter Informationen durch den Benutzer.

Automatische Verfolgung von Ereignissen in der Durchführungsdomäne

Um den Benutzer von der Kommunikationen mit der Leitdomäne zu entlasten und außerdem objektivere Rückmeldungen zu entlasten, verwenden einige Ansätze Techniken zur automatischen Verfolgung von Ereignissen, die in der Durchführungsdomäne beim Umgang mit den Werkzeugen anfallen. Diese Ereignisse werden dann interpretiert und fließen in die Prozessmodellinterpretation ein.

User Interface Places in SLANG

In der SPADE-Umgebung [BBFL94] werden, wie bereits oben beschrieben, Nachrichten, die von den Werkzeugen der DEC Fuse-Umgebung an den Message Server abgesetzt werden, über das SPADE Communication Interface (SCI) abgefangen, als Token reifiziert und in *user interface places* der aktuell interpretierten SLANG-Netze abgebildet.

Event Monitoring in Provence

Auf einer systemtechnisch noch tieferen Ebene als bei SPADE setzt das Ereignis-Monitoring in der prozesszentrierten Umgebung *Provence* [BaKr93; BaKr95] an. Hier werden auf Betriebssystemebene Dienste des *n-Dimensional File System* (n-DFS) [FoKR94] genutzt, um Datei- und Verzeichniszugriffe und die Ausführung von Werkzeugen zu überwachen. Diese primitiven Ereignisse werden mit Hilfe von *Yeast* [KrRo95], einem System zur Spezifikation und Auswertung von Ereignis-Aktions-Regeln, zu komplexeren Ereignissen aggregiert und der Leitdomäne zur Verfügung gestellt. Problematisch an diesem Ansatz ist, dass Ereignisse vom Betriebssystem-Niveau auf die semantische sehr viel höhere Ebene der Prozessmodellierung abgebildet werden müssen. So ist zum Beispiel aus den verfügbaren Publikationen nicht ersichtlich, wie aus dem Speichern einer Quellcode-Datei Rückschlüsse auf den aktuellen Arbeitsprozesszustand (z.B. Beendigung einer Fehlerkorrekturaufgabe) gezogen werden können, zumal als Kontextinformationen für die Ereignisbeschreibung nur der Identifier (`pid`) des ereignis-auslösenden Betriebssystemprozesses, der Name des Besitzers des Betriebssystemprozesses (`uid`) sowie der Name des Rechners (`host`), auf dem das Ereignis ausgelöst wurde, zur Verfügung stehen.

3.3.5.3 Fazit

Eine prozessintegrierte Umgebung kann nur solange sinnvolle Unterstützungsleistungen liefern, wie das Prozessmodell in der Modellierungsdomäne, der beobachtete Prozess in der Leitdomäne und der reale Prozess in der Durchführungs-

domäne miteinander im Einklang stehen. Insbesondere sollten Beobachtungsinkonsistenzen und unbewusste Modellabweichungen vermieden werden.

Der Informationsaustausch zwischen den Prozessdomänen muss daher durch ein explizites Synchronisationsprotokoll geregelt werden, das zwischen reaktivem und proaktivem Unterstützungsmodus unterscheidet, Rückmeldungen bezüglich automatisch ausgeführter Aktionen ebenso wie Benutzerauswahlen berücksichtigt und Prozessabweichungen oder den Abbruch der Prozessunterstützung geeignet behandelt.

In den meisten existierenden prozesszentrierten Umgebungen nimmt die Leitdomäne Rückmeldungen direkt vom Benutzer über spezielle Assistenzwerkzeuge entgegen. Diese Art der Interaktion ist für den Benutzer insbesondere bei feingranularen Prozessen sehr mühsam und fehlerträchtig. Einige Ansätze wie SPADE und Provence umgehen dieses Problem, indem sie aus direkt oder indirekt von den Werkzeugen ausgelösten Ereignissen Rückschlüsse auf die Prozessdurchführung zu ziehen versuchen. Diese Lösung bewegt sich jedoch auf logisch niedrigem Niveau. Außerdem gilt auch hier, dass die Interaktion zwischen der Prozessmaschine und den Werkzeugen keinem geregelten Protokoll folgt.

3.3.6 Prozesssensitive Benutzeroberflächen

3.3.6.1 Motivation

Bei der Beurteilung der Präsentationsintegration werden üblicherweise zwei Aspekte unterschieden: das *visuelle Erscheinungsbild und Verhalten* der Werkzeuge sowie die verwendeten *Interaktionsparadigmen* [ThNe92; Sche93].

Visuelles Erscheinungsbild

Die Uniformität des visuellen Erscheinungsbilds fängt bei der Gestaltung von Grundelementen der grafischen Benutzeroberfläche an. Dazu gehören z.B. einheitliche Fensterrahmen, Schaltflächen, Auswahllisten etc. Von einer präsentationsintegrierten Umgebung wird man weiterhin erwarten, dass semantisch vergleichbare Funktionen unterschiedlicher Werkzeuge konsistent benannt werden (z.B. „Speichern“ vs. „Sichern“), identische Parameterlisten erwarten, an ähnlichen Positionen innerhalb der Menühierarchie angeordnet sind und über gleiche Tastaturkürzel angesprochen werden können. Beispielsweise hat sich eingebürgert, dass im jeweils letzten Pull-down-Menü eines Fensters Hilfefunktionen zu finden sind, die sich auch über die Funktionstasten F1 (Hilfeübersicht) und Crtl-F1 (kontextsensitive Hilfe) aktivieren lassen. In den Bereich des so genannten „look & feel“ fällt auch die Vereinheitlichung des Verhaltens der GUI-Elemente, z.B. die Semantik einer „Drag&Drop“-Operation, die Reaktion auf einen Mausklick mit der linken Taste (Objektselektion) bzw. rechten Taste (Aufklappen eines kontextsensitives Menüs) oder die konsistente Verwendung der Zwischenablage (Clipboard). In einem etwas breiteren Sinne verlangt man auch, dass ähnliche Interaktionen in unterschiedlichen Werkzeugen annähernd gleiche Antwortzeiten bedingen. Im Rahmen dieser Arbeit setzen wir eine einheitliche Gestaltung des visuellen Erscheinungsbilds stillschweigend voraus, da hier insbesondere im Desktop-Bereich durch die Marktdominanz von Microsoft und durch die Etablierung softwareergonomischer Gestaltungsrichtlinien [Micr95] bereits weitreichende Fortschritte im Vergleich zu früheren Jahren erzielt worden sind.

Interaktionsparadigmen

Aus Sicht der Prozessintegration interessanter ist die Frage des verwendeten *Interaktionsparadigmas*. Das Interaktionsparadigma bestimmt die Art und Weise, wie der Benutzer auf die Entwurfsobjekte und Bearbeitungsfunktionen zugreift. Traditionell werden das werkzeug-/funktionsorientierte oder dokumenten-/objektorientierte Interaktionsparadigma unterschieden [Balz96; Wand93].

Werkzeug-/ Funktionsorientiert

Beim werkzeug-/funktionsorientierten Ansatz aktiviert der Benutzer zunächst die für einen Arbeitsschritt benötigte Funktion bzw. ein Werkzeug (mit seinen Funktionen) und bestimmt dann, auf welchem Objekt oder welchen Objekten die Funktion angewandt werden soll. Der Benutzer muss also wissen, welche Werkzeuge die seinen Arbeitsschritt unterstützenden Funktionen anbieten und wie in diesen Werkzeugen der Zugriff auf die zu bearbeitenden Dokumente oder Objekte erfolgt.

Dokument-/ Objektorientiert

Charakteristisch für das dokumenten- bzw. objektorientierte Paradigma ist, dass der Benutzer zunächst das zu bearbeitende Objekt auswählt. Aus dem Typ und Zustand des Objekts ergeben sich dann die möglichen Bearbeitungsfunktionen, die dem Objekt als Methoden zugeordnet sind. Für den Benutzer tritt die Frage, von welchem Werkzeug eine bestimmte Funktion abgeboten wird, in den Hintergrund, so dass er sich nicht mehr darum kümmern muss, das geeignete Werkzeug auszuwählen und zu aktivieren. Das Hinzufügen neuer Werkzeuge in die Arbeitsumgebung resultiert aus Benutzersicht in dem Erweitern des Angebotsspektrums an verfügbaren Objekttypen und darauf ausführbaren Methoden.

Prozesssensitives Interaktionsparadigma

In Verbindung mit einer Prozesssteuerung kann ein *prozesssensitives Interaktionsparadigma* verwirklicht werden. Hierbei passen sich die Interaktionsmöglichkeiten in einem Werkzeug dynamisch dem aktuellen Zustand der Prozessmodellausführung an. Dies bedeutet, dass in der Werkzeugoberfläche der Zugriff auf die für den aktuellen Prozesskontext irrelevante Funktionen und Objekte eingeschränkt wird (z.B. durch Ausblenden von Menüpunkten oder durch Deaktivieren der Selektierbarkeit von Objekten). Weiterhin sollte das Werkzeug in der Lage sein, Objekte, auf die sich ein aktuell von der Leitdomäne angeforderten Prozessschritt bezieht, visuell hervorzuheben, um die Aufmerksamkeit des Benutzers auf diese Objekte zu lenken.

Die Realisierung eines prozesssensitiven Interaktionsparadigmas leistet einen wichtigen Beitrag zur Vermeidung von Abweichungen und Inkonsistenzen zwischen den Prozessdomänen und steht somit in engem Zusammenhang mit der Synchronisation zwischen den Prozessdomänen (siehe auch Abschnitt 3.3.5). Da dem Benutzer standardmäßig nur die gemäß Prozessmodell und aktuellem Ausführungszustand gültigen Funktionen und Objekte zur Auswahl angeboten, reflektiert das Verhalten der Werkzeuge stets den aktuellen Zustand der Leitdomäne wider. Die prozesssensitive Anpassung der Werkzeugoberflächen verkörpert also den Synchronisationsschritt *von* der Leitdomäne *zur* Durchführungsdomäne (im Gegensatz zur den im vorigen Abschnitt betrachteten *Rückmeldungen* aus der Durchführungsdomäne zur Leitdomäne) und vermindert somit die Gefahr von Beobachtungsabweichungen. Indirekt werden damit auch *unbewusste* Modellabweichungen verhindert (also das Verletzen von Vorgaben des Prozessmodells). Allerdings sollten die Werkzeuge auf explizite Anforderung vom Benutzer den gesamten Funktionsvorrat zu Verfügung stellen und somit bewusste Prozessabweichungen zulassen.

3.3.6.2 Bewertung existierender Ansätze

Aus Benutzersicht weist das dokumentenorientierte Interaktionsparadigma, das in den letzten zehn Jahren besonders im Umfeld Desktop-zentrierter Betriebssysteme (Windows, MacOS) starke Verbreitung gefunden hat, viele Parallelen zu einem prozessorientiertem Paradigma auf. Wie oben bereits angedeutet, treten hierbei die eigentlichen Bearbeitungsfunktionen der Werkzeuge in den Hintergrund. Das heißt insbesondere, dass die Funktionen nicht mehr in Form expliziter Werkzeuge erscheinen, sondern sich beispielsweise in kontextsensitiven Menüs manifestieren, die immer nur dann in Erscheinung treten, wenn sie sinnvoll eingesetzt bzw. benötigt werden. Dies führt dann soweit, dass ein Dokument zum „intelligenten Assistenten seines eigenen Bearbeiters" wird [Grif98]. Dokumentenmodelle wie z.B. COM/OLE [Broc95] oder OpenDoc [OrHE96] liefern das technische Fundament, um Dokumentobjekte verschiedener Werkzeuge auch visuell innerhalb eines Fensters als so genannte Compound Documents ineinander schachteln zu können. Dadurch verwischen die Grenzen zwischen einzelnen Werkzeugen, so dass aus Benutzersicht eine bestimmte Funktion nicht länger einem Werkzeug, sondern einem Dokumentobjekt zugeordnet wird. Im Unterschied zu einem prozesssensitiven Interaktionsparadigma beruht die Kontextsensitivität des Funktionsangebots jedoch allein auf der aktuellen Objektselektion (und eventuell dem spezifischen Objektzustand) und unterliegt nicht irgend welchen explizit definierten Prozessmodellen bzw. der Prozessmodellausführung in der Leitdomäne.

Kontextsensitivität in Compound Documents

Ansätze aus dem Bereich der prozesszentrierten Umgebungen/ Workflowmanagementsysteme haben bislang die Auswirkungen der Prozessorientierung auf das Interaktionsparadigma in den verwendeten Werkzeugen und Applikationen weitgehend vernachlässigt. Der Zugriff auf Objekte und Funktionen in den Werkzeugen kann nicht oder nur grobgranular (meist bezogen auf die Daten) von der Prozessmodellausführung beeinflusst werden, woraus schwerwiegende Synchronisationsprobleme resultieren [Böhm98; BeMü99; Schm96; Schr97].

Eine interessante Ausnahme bildet die im Rahmen des *Eureka Software Factory*-Projekts entwickelte Umgebung *East* [Simm91; Simm93]. In East beschreiben als Task Template bezeichnete Prozessmodelle nicht nur die zu bearbeitende Aufgabe, sondern auch die charakteristischen Eigenschaften des Daten- und Funktionszugriffs in der Werkzeugumgebung. Beim Einloggen in die East-Umgebung wählt der Benutzer zunächst eine Task zur Bearbeitung aus. Die Task-Selektion bilden den Ausgangspunkt für die Etablierung einen *Task-spezifischen Sub-Umgebung*, in der der Entwickler nur noch den Zugang zu den Daten und Operationen erhält, die zur Erfüllung der Aufgabe notwendig sind.

Aufgabenspezifische Umgebungen in East

Eine Task-spezifische Sub-Umgebung besteht aus zwei Komponenten: einer *Daten-Umgebung*, die alle Objekte der Objektbasis beinhaltet, auf die der Benutzer im Rahmen der ausgewählten Task Zugriff hat, und einer *Dienst-Umgebung*, die eine hinreichende Untermenge aller möglichen Operationen umfasst. Die Definition der Daten-Umgebung erfolgt über Sichtbarkeitsmechanismen des zugrunde liegenden PCTE Objektmanagementsystems [WaJo93]. Für jedes Werkzeug und jedes Task Template lassen sich Arbeitsschemata, d.h. eingeschränkte Sichten auf das Globalschema, definieren. Die Benutzerschnittstelle aller Werkzeuge wird über das East User Interface Management System (UIMS) organisiert, wodurch die Präsentation und der Benutzerdialog von der eigentlichen Funktionalität des Werkzeugs abgekoppelt wird. Eingeschränkt durch den aktuellen Prozesskontext,

unterstützt das East-UIMS einen objektorientierten Interaktionsstil, d.h. die in einem Fenster dargestellten Objekte sind mit einem Pop-up-Menü assoziiert, das alle auf einem Objekt anwendbaren Operationen enthält, die durchaus von verschiedenen Werkzeugen stammen können. East bietet somit eine prozesssensitive Anpassung der Benutzeroberfläche der Werkzeugumgebung. Allerdings beschreiben Tasks eher gröbere Arbeitseinheiten und sind nicht zur feingranularen Prozessmodellierung gedacht. Daher ändert sich der verfügbare Vorrat an zugreifbaren Daten und ausführbaren Operationen auch nicht mehr, nachdem sich der Benutzer nach Auswahl einer Aufgabe in die East-Umgebung eingeloggt hat.

3.3.6.3 Fazit

Beim prozesssensitiven Interaktionsparadigma hängt der dem Benutzer aktuell zur Verfügung stehende Funktionsvorrat anders als beim objektorientierten Paradigma nicht nur vom ausgewählten Dokumentkontext, sondern auch von der Prozessausführung ab. Dadurch wird der intendierte Prozess für den Benutzer direkt in seinen Werkzeugen sichtbar. Eine Überfrachtung mit irrelevanter Funktionalität wird vermieden, und die Gefahr unbewusster Prozessabweichungen wird reduziert.

Die uns bekannten prozesszentrierten Umgebungen und Workflow-Managementsystem bieten keine Unterstützung für die systematische Etablierung prozesssensitiver Benutzeroberflächen bei den eingebundenen Werkzeugen und Applikationen. Lediglich in der East-Umgebung existiert das Konzept aufgabenspezifisch angepasster Werkzeugumgebungen, allerdings bezogen auf jeweils relativ grobgranulare Arbeitseinheiten.

Die Idee prozesssensitiver Benutzeroberflächen wird auch von Assistenten und Interface-Agenten mithilfe einer stark aufgabenbezogenen Dialogsteuerung verwirklicht. Wie bereits in Abschnitt 2.2.3 diskutiert, bieten diese Systeme jedoch keinerlei Anpassbarkeit an Prozessänderungen.

3.3.7 Werkzeugunterstützter Aufruf von Prozessfragmenten

3.3.7.1 Motivation

In Abschnitt 3.3.5 hatten wir bereits darauf hingewiesen, dass aus einer nur fragmentarischen Modellierung von Prozessen zwei unterschiedliche Ausführungsmodi resultieren: im proaktiven Modus kontrolliert die Leitdomäne das Geschehen in der Durchführungsdomäne, während im reaktiven Modus kein definiertes Prozessfragment für die Unterstützung des aktuellen Prozesses vorliegt und die Leitdomäne somit inaktiv ist.

Übergang vom reaktiven in den proaktiven Unterstützungsmodus

Für den Übergang vom reaktiven zum proaktiven Modus der aktuelle Prozessstatus in der Durchführungsdomäne mit den Eintrittsbedingungen der definierten Prozessfragmente verglichen werden. Da sich gerade in den Werkzeugen der aktuelle Prozesszustand manifestiert, sollten die Werkzeuge den Benutzer bei der Suche nach anwendbaren Prozessfragmenten aktiv unterstützen und die aktuell anwendbaren Prozessfragmente in der Benutzeroberfläche zur Auswahl (z.B. über Menüpunkte) anbieten. Im Idealfall sollte es daher für den Benutzer transparent

sein, ob er eine werkzeugeigene Funktionalität aktiviert oder die Ausführung eines definierten Prozessfragments in der Leitdomäne anstößt.

Da der Vorrat an definierten Prozessfragmenten in der Modellierungsdomäne sich mit der Zeit ändert, darf die Erkennung und Darstellung anwendbarer Prozessfragmente nicht in den Werkzeugen hartkodiert werden. Daher müssen die Werkzeuge auf die aktuellen Prozessdefinitionen zugreifen können und über eine entsprechend flexible Kommandoverwaltung (z.B. erweiterbare Menüstrukturen) verfügen.

3.3.7.2 Bewertung existierender Ansätze

Die Integrationsstrategien in existierenden prozesszentrierten Umgebung und Wokflow-Managementsystemen sehen für die dort eingebundenen Werkzeuge und Applikationen keine aktive Rolle bei der Aktivierung von Prozessfragmenten vor. Vielmehr läuft die Interaktion zwischen der Leitdomäne (Prozessmaschine) und der Durchführungsdomäne (Werkzeuge) in der Regel nach dem Client-Server-Prinzip ab. Hierbei haben die Werkzeuge als reine Diensterbringer keinerlei Kenntnis der Prozessdefinitionen und des aktuellen Zustands der Prozessmodellausführung und können folglich auch nicht die Ausführung von Prozessfragmenten initiieren. Stattdessen erfolgt die Aktivierung von Teilprozessen entweder explizit durch den Benutzer über spezielle Benutzerschnittstellen (Agenda-Manager, Arbeitskontexte o.ä.) oder implizit durch das bereits in Abschnitt 3.3.5 diskutierte Abhorchen von Ereignissen in der Durchführungsdomäne wie z.B. in Provence und SPADE.

Die WfMC sieht in ihrer *Workflow Management API Interface 2&3 Specification* [WFMC98a] eine Schnittstelle *zum* Workflow-System vor, über die so genannte *workflow-enabled applications* Informationen über anwendbare Prozessfragmente oder aktuell laufende Aktivitätsinstanzen erfragen können. Gedacht ist diese Schnittstelle allerdings weniger für die eigentlichen Werkzeuge einer Entwurfsumgebung, sondern eher für prozesszentrierte Administrationswerkzeuge wie Agenda-Manager, Projektmanagement-Werkzeuge und Monitoring-Werkzeuge.

Moderne Entwicklungsumgebungen, insbesondere aus dem Windows-Umfeld, bieten Mechanismen zur *ad-hoc*-Erweiterung des Funktionsumfangs durch den Benutzer oder durch Dritthersteller. Als Beispiele sind die Möglichkeiten zur Makroprogrammierung in der Microsoft-Office-Familie oder die Einbindung von so genannten *Add-ins* in Umgebungen wie Microsoft Visual Studio oder Rational Rose zu nennen. Diese Mechanismen, die sich intern meist auf COM-Schnittstellen des zugrunde liegenden Objektmodells des Werkzeugs abstützen, stellen einen grundsätzlichen Ansatzpunkt für die werkzeugseitige Integration mit einem Prozessausführungsmechanismus bereit. Es ist uns allerdings kein *systematischer* Ansatz für eine prozessorientierte Nutzung solcher Erweiterungsschnittstellen bekannt.

3.3.7.3 Fazit

Der Übergang vom reaktiven in den proaktiven Unterstützungsmodus ist dadurch charakterisiert, dass die Durchführungsdomäne Prozessunterstützung von der Leitdomäne anfordert. In diesem Zusammenhang bedeutet Prozessintegration, dass die Werkzeuge der Durchführungsdomäne hierbei eine aktive Rolle spielen und als Clients der Leitdomäne fungieren. Aus Benutzersicht sollte kein Unter-

schied zwischen der Aktivierung einer werkzeugeigenen Funktion und eines extern definierten Prozessfragments bestehen.

Die heute üblichen Integrationsstrategien in prozesszentrierten Umgebungen vernachlässigen diesen Aspekt weitgehend und sehen Werkzeuge lediglich in der Rolle eines Diensterbringers ohne Kenntnis der zu unterstützenden Prozesse. Schnittstellenspezifikationen der WfMC für *workflow-enabled applications* und Erweiterungsmechanismen in modernen Werkzeugen können aber eine Grundlage für eine aktivere Rolle der Werkzeuge bieten.

3.4 Fazit

In dieses Kapitel sind wir mit der Kernhypothese gegangen, dass die explizite Modellierung von Prozessen in prozesszentrierten Entwicklungsumgebungen eine gute Grundlage für die adaptable und kontextsensitive Prozessunterstützung von Entwickleraktivitäten darstellt. Der Fokus solcher Ansätze lag in der Vergangenheit jedoch stark auf den zugrunde liegenden Prozessmodellierungssprachen und Ausführungsmechanismen, während die Auswirkungen auf die interaktiven Werkzeuge der Entwicklungsumgebung weitgehend ignoriert wurden. Ziel des Kapitels war es daher, Voraussetzungen und Anforderungen an eine engere Integration zwischen der Modellierungs- und Leitdomäne einerseits und der Durchführungsdomäne andererseits herauszuarbeiten.

Zur Strukturierung der unterschiedlichen Integrationsaspekte haben wir zunächst allgemeine Klassifikationsansätze für die Integration von Entwicklungsumgebungen aus der Literatur vorgestellt. Die Diskussion ergab, dass der Begriff der Prozessintegration bereits in einer Reihe von Publikationen aufgegriffen wird und als ein Adaptionsproblem unter Nutzung der Dienste „tieferer“ Integrationsdimensionen (Daten, Kontrolle, Präsentation) verstanden wird. Welche konkreten Anforderungen sich aus einer prozessorientierten Sichtweise an die Integration *zwischen* den Prozessdomänen ergeben, gilt jedoch noch als weitgehend unverstanden und wird in den zitierten Publikationen nicht im Detail erörtert.

Um die Konsequenzen einer engeren Integration der Prozessdomänen näher zu beleuchten, haben wir sechs zentrale Integrationsanforderungen hergeleitet:

- Datenintegration zwischen den Prozessdomänen;
- Prozessorientierte Mediation von Werkzeuginteraktionen;
- Konzeptuelle Modellierung von Werkzeugen;
- Synchronisation zwischen Leit- und Durchführungsdomäne;
- Prozesssensitive Anpassung der Interaktion in den Werkzeugen;
- Werkzeugunterstützter Aufruf von Prozessfragmenten.

Die Betrachtung existierender prozesszentrierter Umgebungen sowie weiterer Ansätze aus den Bereichen Datenintegration, Kommunikationsinfrastrukturen, komponentenbasierte Softwareentwicklung, Werkzeugspezifikation, Prozessmodellierung, Benutzeroberflächen und Softwareergonomie ergab, dass zu Teilaspekten bereits mitunter ausgereifte Lösungsansätze vorliegen. Uns ist jedoch *keine* prozesszentrierte Umgebung bekannt, die alle genannten Anforderungen in einem ganzheitlichen Ansatz zusammenführt und umsetzt.

Teil 2

Lösungskonzept



Überblick über den Lösungsansatz

In diesem Kapitel geben wir einen kurzen Überblick über den in dieser Arbeit vorgestellten PRIME-Ansatz für prozessintegrierte Werkzeuge, der Lösungen zu den im vorigen Kapitel angesprochenen Problemen bietet. PRIME basiert auf folgenden Kernelementen, die in den Kapiteln 5 – 7 genauer dargestellt werden.

- *Integrierte Prozess- und Werkzeugmodellierung:*

  In PRIME werden Prozesse und Werkzeuge gleichberechtigt auf einer konzeptuellen Ebene repräsentiert. Die Integration von Prozess- und Werkzeugmodellen zu einem so genannten *Umgebungsmodell* bildet die Grundlage für:

  - Datenintegration durch die Definition gemeinsamer Produktdatenschemata für Werkzeuge und Prozessmaschine;
  - Prozessmediierte Werkzeuginteraktionen durch Separierung von Prozess- und Werkzeugaspekten in den jeweiligen Teilmodellen;
  - Uniforme Beschreibung von Werkzeugdiensten und Prozessfragmenten;
  - Synchronisation durch eine explizite Definition der erwarteten Rückmeldungen an die Leitdomäne;
  - Werkzeugunterstützter Aufruf von Prozessfragmenten;
  - Dynamische Anpassung der in den Werkzeugen zugreifbaren Objekte und Dienste gemäß Prozessmodell und aktuellem Prozesszustand.

  Die Metamodelle für die Prozess- und Werkzeugmodellierung werden in Kapitel 5 vorgestellt.

- *Flexible Ablaufmodellierung*

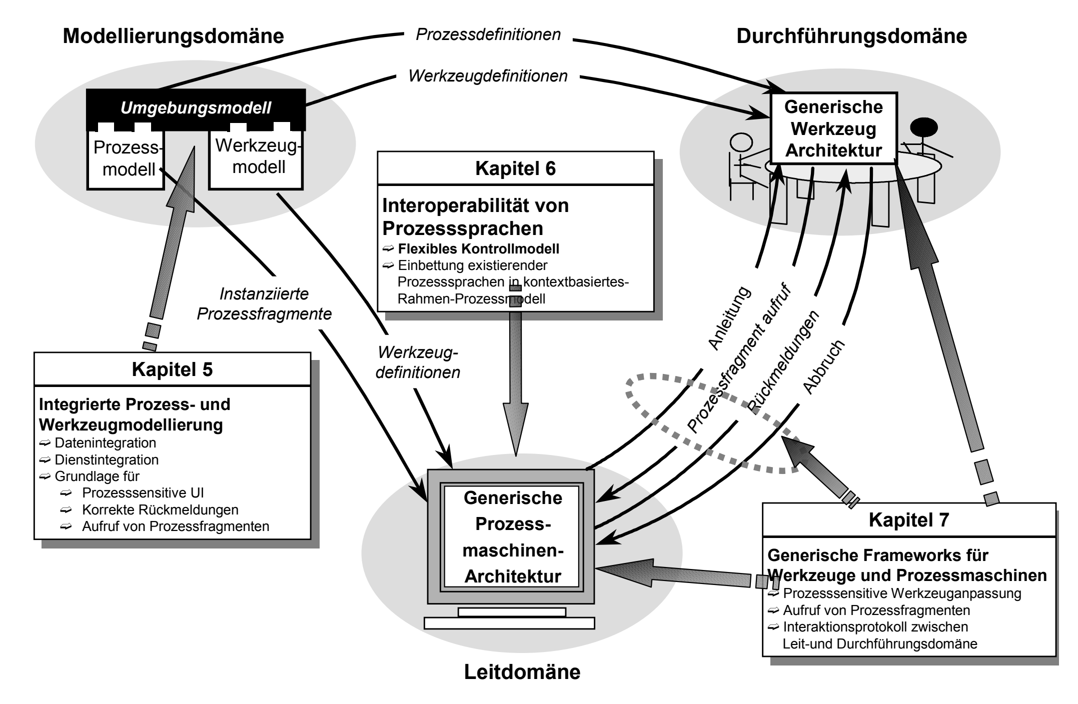
  Prozessfragmente können in PRIME ausgehend von elementaren Werkzeugdiensten zu komplexen, kontextabhängigen Ablaufplänen aggregiert werden. Je nach Modellierungsanforderungen sind unterschiedliche Formalismen zur Kontrollflussspezifikation innerhalb eines Teilfragments besonders geeignet, z.B. Petrinetze oder endliche Automaten. In Kapitel 6 diskutieren wir ein Konzept für die modellierungsseitige Interoperabilität zwischen verschiedensprachlich spezifizierten Prozessfragmenten, das auf dem Prinzip der Schnittstellentransformation in komponentenbasierten Ansätzen basiert.

- *Integrierte Frameworks für Leit- und Durchführungsdomäne*

  Basierend auf dem integrierten Umgebungsmodell entwickeln wir in Kapitel 7 generische Architekturen für prozessintegrierte Werkzeuge und Prozessmaschinen. Realisiert werden beide Architekturen in Form objektorientierter Implementierungs-Frameworks, die als Kernkomponenten Interpreter für die werkzeug- bzw. prozessrelevanten Anteile des Umgebungsmodells beinhalten. Das dynamische Zusammenspiel zwischen den Werkzeugen in der Durchführungsdomäne und der Prozessmaschine in der Leitdomäne wird durch ein umfassendes Interaktionsprotokoll geregelt.

Der Unterschied zwischen einer prozess*zentrierten* Umgebung wie z.B. SPADE [BaDF96] oder Dynamite [HJKW96] und einer prozess*integrierten* Umgebung nach dem PRIME-Ansatz lässt sich gut anhand der in Abb. 18 dargestellten Erweiterungen des Domänenmodells von Dowson charakterisieren (vergleiche auch Abb. 4 auf Seite 35). In der Modellierungsdomäne ist neben dem Prozessmodell auch das damit integrierte Werkzeugmodell angesiedelt. Die Interpretation des Werkzeugmodells versetzt die Werkzeuge in die Lage, ihr Verhalten dynamisch an den aktuellen Prozesszustand anzupassen und die Ausführung von Prozessfragmenten von der Leitdomäne anzufordern. Insgesamt kommt den Werkzeugen in der Durchführungsdomäne somit eine wesentlich aktivere Rolle bei der Prozessunterstützung zu, die sich auch in erweiterten Interaktionen zwischen der Leit- und Durchführungsdomäne widerspiegelt.

***Abb. 18**: Erweiterung der Domänenmodells von Dowson in einer prozessintegrierten Umgebung*





Integrierte Prozess- und Werkzeugmodelle

Im voran gegangenen Überblickskapitel haben wir argumentiert, dass die explizite Modellierung von Prozessen und Werkzeugen auf der gleichen konzeptuellen Ebene eine der tragenden Säulen unseres Konzepts für die Prozessintegration von Werkzeugen darstellt. Diesen Grundgedanken wollen wir nun vertiefen und geeignete Metamodelle, d.h. Sprachen, zur integrierten Definition von Prozessfragmenten und Werkzeugen vorstellen.

In Abschnitt 5.1 geben wir als Vorbereitung zunächst einen kurzen Überblick über die Notationen, die wir in diesem und den nachfolgenden Kapiteln zur Darstellung der unterschiedlichen Modelle und Metamodelle verwenden werden. Bei der Definition des integrierten Prozess- und Werkzeugmodells gehen wir schrittweise vor. In Abschnitt 5.2 grenzen wir auf Basis der in Kapitel 3 aufgestellten Anforderungen die Gegenstandsbereiche des Prozess- und des Werkzeugmodells voneinander ab und arbeiten auf informaler Ebene die inhaltlichen Querbezüge zwischen den beiden Modellen heraus. In Abschnitt 5.3 stellen wir das NATURE-Prozessmetamodell vor, das uns als Ausgangspunkt für die kontextbasierte Definition von Prozessfragmenten dient. Abschnitt 5.4 behandelt die in dieser Arbeit entwickelte Erweiterung des NATURE-Prozessmodells um Konzepte zur Werkzeugmodellierung. Die Integration des Prozess- und des Werkzeugmetamodells innerhalb des so genannten Umgebungsmetamodells wird in Abschnitt 5.5 beschrieben. Abschnitt 5.6 illustriert die Verwendung der Metamodelle anhand eines kleinen Beispiels und Abschnitt 5.7 fasst die wesentlichen Beiträge dieses Kapitels zusammen.

5.1 Darstellung der Modelle

Für die Darstellung der in diesem und den nachfolgenden Kapitel vorgestellten konzeptuellen Modelle und Metamodelle benötigen wir geeignete Formalismen. Hierbei stützen wir uns auf allgemein etablierte Basiskonzepte, Strukturierungs- und Abstraktionsprinzipien des objektorientierten Paradigmas ab und notieren unsere Modelle als UML-Klassendiagramme und formalisieren sie, sofern erforderlich und hilfreich, zusätzlich mit Hilfe der logikbasierten Modellierungssprache O-Telos.

5.1.1 UML

Zur übersichtsartigen Darstellung der Modelle verwenden wir Klassendiagramme der Unified Modeling Language UML [BoJR99]. Die Wahl der UML als primär in dieser Arbeit verwendeter Notation liegt zum einen darin begründet, dass die UML in einer intuitiven, grafischen Darstellung alle wesentlichen Konzepte bereitstellt, wie sie etwa in Arbeiten zur Standardisierung von Objektmodellen [SoKe95], zu objektorientierten Programmiersprachen [CaWe85; Meye90; AbCA96] oder zu objektorientierten Datenbanken [Atk*89; Beer90; LaVo97] gefordert werden. Zum anderen stellt die UML als Resultat der Vereinigung und Konsolidierung der populärsten Modellierungsnotationen der 80er und frühen 90er Jahre (Rumbaugh's OMT [Rum*91], Booch's OOAD [Booc94] und Jacobson's OOSE [JCJÖ92]) einen (vorläufigen) Endpunkt im Bereich der konzeptuellen, objektorientierten Sprachen dar und hat mit der Standardisierung durch die Object Management Group [OMG#97a; Kobr99] den Status einer objektorientierten „lingua franca" erreicht. Die Notation und Semantik der in dieser Arbeit verwendeten UML-Modellierungselemente wird als bekannt vorausgesetzt. Weitergehende Informationen sind der UML-Sprachdefinition in [BoJR99] bzw. [RuJB99] sowie diversen UML-Lehrbüchern (z.B. [ErPe98; HiKa99]) zu entnehmen

Objektorientierte "lingua franca"

5.1.2 O-Telos

Für eine Formalisierung der in der Arbeit vorgestellten konzeptuellen Modelle wird die logikbasierte, objektorientierte Sprache O-Telos [Jeus92; MBJK90] verwendet. Der Grund für die Zuhilfenahme einer weiteren konzeptuellen Modellierungssprache neben UML liegt in der Tatsache, dass wir über die Ausdrucksmöglichkeiten von UML-Klassendiagrammen hinaus die Semantik der konzeptuellen Modelle durch Angabe von Integritätsbedingungen und Regeln weiter präzisieren wollen. Die von den UML-Klassendiagrammen bereitgestellten Ausdrucksmittel (im Wesentlichen Kardinalitätseinschränkungen, vordefinierte Abhängigkeitskategorien, Stereotype, informelle Notizen) sind hierfür nicht mächtig genug bzw. im Metamodell der UML nicht hinreichend formal fundiert. Die UML stellt zwar mit der Object Constraint Language [OMG#97d] eine eigene Spezifikationssprache bereit, in der jedoch im Gegensatz zu O-Telos u.a. keine Regeln über mehr als eine Instanziierungsebene formuliert werden können. Außerdem steht mit dem deduktiven Objektmanager ConceptBase [Jar*95] eine operationale O-Telos-Implementierung zur Verfügung, die durch die automatische Überprüfung von Basisaxiomen und benutzerdefinierten Integritätsbedingungen sowie durch Deduktion intensionalen Wissens aus Regeln und Anfragen die Erstellung konsistenter konzeptueller Modelle erheblich erleichtert.

Die Frame-basierte Notation von O-Telos wird im Folgenden als bekannt vorausgesetzt[16]. Aus Gründen der Übersichtlichkeit werden wir zur Darstellung der Modelle hauptsächlich grafische UML-Diagramme verwenden und aus Platzgründen nicht für jedes UML-Konzept die korrespondierenden O-Telos-Definitionen aufführen. Vielmehr beschränken wir die Verwendung von O-Telos primär

---

[16] Weitergehende Information zur O-Telos-Sprachdefinition sind in [Jeus92] oder [JaJQ99] zu zu finden.

für die Formulierung spezieller Integritätsbedingungen, Regeln und Anfrageklassen, die im UML-Modell nicht darstellbar sind. Einzelheiten über die modellierungstechnischen Zusammenhänge zwischen der UML und O-Telos sind in [Grun99] zu finden.

5.2 Motivation für integrierte Prozess- und Werkzeugmodelle

Die in Kapitel 3 aufgestellten Anforderungen an eine Integration der Prozessdomänen, insbesondere die in Abschnitt 3.3.4 geforderte *konzeptuelle Beschreibung von Werkzeugdiensten*, die auf der gleichen konzeptuellen Ebene wie die Prozessmodellierung angesiedelt ist, motiviert eine integrierte Betrachtung von Prozessen und Werkzeugen. Bei der Entwicklung eines integrierten Prozess- und Werkzeugmodells lassen wir uns von der Grundüberlegung leiten, dass wir die Kernaspekte der Prozesse und Werkzeuge zunächst unabhängig voneinander in spezifischen Teilmodellen erfassen. Diese Vorgehensweise hat den Vorteil, dass wir in einem ersten Schritt Prozesse unabhängig von einer konkret gegebenen Werkzeugumgebung modellieren können, während wir umgekehrt die Fähigkeiten der zur Verfügung stehenden Werkzeuge prozessneutral beschreiben können. In einem Integrationsschritt werden die Konzepte zur Prozess- und Werkzeugmodellierung dann systematisch zueinander in Bezug gesetzt, so dass wir eine explizite und flexibel anpassbare Zuordnung von Prozessschritten zu Werkzeugfunktionalitäten erhalten.

Separierung von prozessrelevanten und werkzeuginhärenten Aspekten

Die wesentlichen Interdependenzen zwischen Prozess- und Werkzeugmodellen ergeben sich aus der Modellierung der unterschiedlichen Dienste, die der Leit- bzw. Durchführungsdomäne zugeordnet sind und von diesen wechselseitig in Anspruch genommen werden. Gemäß den Anforderungen aus Abschnitt 3.3.4 können drei grundlegende Dienstkategorien in einer prozessintegrierten Entwurfsumgebung unterschieden werden [Poh*99]: *elementare Dienste*, *Beratungsdienste* und *Anleitungsdienste*. Diese Dienstkategorien unterscheiden sich modellierungsseitig u.a. darin, ob sie innerhalb des Prozess- oder des Werkzeugmodells definiert bzw. referenziert werden, und ausführungsseitig, ob die Leitdomäne, d.h. die Prozessmaschine, oder die Durchführungsdomäne, d.h. die Werkzeuge, für ihre Operationalisierung zuständig sind.

- **Elementare Dienste**: Dies sind die Basisaktionen, die von den Werkzeugen einer Entwurfsumgebung zur Verfügung gestellt werden und von der Leitdomäne zur Umsetzung atomarer Prozessschritte angefordert werden. Beispiele für elementare Dienste sind das Kompilieren eines Quelltextes oder das Erzeugen eines Modellbausteins in einem interaktiven Modellierungswerkzeug, z.B. einem Entity-Relationship-Editor (ER-Editor). Für die Ausführung elementarer Dienste sind grundsätzlich die Werkzeuge der Durchführungsdomäne zuständig. Die definierende Schnittstellenbeschreibung elementarer Dienste ist Bestandteil des Werkzeugmodells. Umgekehrt werden elementare Dienste als atomare Prozessschritte im Prozessmodell referenziert.

- **Beratungsdienste**: Beratungsdienste entsprechen spezifischen Arbeitsmodi, in denen der Zugriff auf die im aktuellen Prozesskontext relevanten bzw. erlaubten Dienste und Produkte eingeschränkt ist. Beratungsdienste unterstützen den Entwickler somit bei der Entscheidung unter den mögli-

chen alternativen Vorgehensweisen. Beratungsdienste werden logisch als Teil des Prozessmodells, d.h. in der Modellierungsdomäne, definiert. Die *Ausführung* eines Beratungsdienstes hat jedoch Auswirkungen auf das Verhalten der Werkzeuge in der Durchführungsdomäne und liegt daher in der Verantwortung der Werkzeuge. Nach Anforderungen durch die Prozessmaschine müssen die Werkzeuge die an der Benutzeroberfläche angebotenen Kommandos und die auswählbaren Produkte gemäß der Definition des Beratungsdienstes und der aktuell relevanten Produktinstanzen anpassen (siehe Anforderung „Prozesssensitive Benutzeroberfläche" in Abschnitt 3.3.6). Als Beispiel für einen Beratungsdienst betrachten wir die Verfeinerung eines Entitätstypen als Teil eines Prozessmodells für die Informationssystem-Modellierung mit Hilfe der Entity-Relationship-Methode (ER). Als im aktuellen Kontext erlaubte Alternativen seien zwei alternative Vorgehensweisen im Prozessmodell vorgesehen: das Hinzufügen eines Diskriminatorattributes oder die Spezialisierung des Entitätstypen. Sobald dieser Beratungsdienst von der Prozessmaschine aktiviert wird, muss das entsprechende ER-Modellierungswerkzeug die definierten Alternativen dem Benutzer anbieten (z.B. als Menükommandos) und alle andere Optionen, etwa die Partitionierung des Entitätstypen in zwei unabhängige Entitätstypen, ausblenden.

- **Anleitungsdienste**: Darunter verstehen wir die Ausführung von Prozessfragmenten, die innerhalb des Prozessmodells definiert werden. Anleitungsdienste definieren Strategien, die in der Regel mehrere Werkzeuge involvieren, in denen bestimmte Schritte durch Ausführungsdienste automatisiert werden oder der Benutzer durch Beratungsdienste in der Auswahl des nächsten Schritts unterstützt wird. Daher ist es nicht sinnvoll, die Ausführung von Anleitungsdiensten einem bestimmten Werkzeug zuzuordnen. Vielmehr hat die Ausführung von Anleitungsdiensten als Abfolge von Teilschritten, die wiederum durch bestimmte Dienste umgesetzt werden, koordinierenden Charakter und liegt in der Zuständigkeit der Leitdomäne, d.h. der Prozessmaschine. Gemäß der Anforderung A6 („Werkzeugunterstützter Aufruf von Prozessfragmenten", siehe Abschnitt 3.3.7) müssen jedoch Anleitungsdienste aus den Werkzeugen der Durchführungsdomäne aktiviert werden können. Als Beispiel für einen Anleitungsdienst sei das Prozessfragment „Spezialisierung eines Entitätstypen" angeführt, das aus einer Abfolge mehrerer Schritte in unterschiedlichen Werkzeugen besteht. Nach Aktivierung dieses Prozessfragments im ER-Editor lenkt und kontrolliert die Prozessmaschine die Durchführung des Spezialisierungsvorgangs gemäß der im Prozessmodell definierten Ablaufdefinition.

Zusammenfassend lässt sich feststellen, dass Informationen über den Anwendungskontext von Diensten sowie das Wissen über Vorgehensauswahlen (Beratungsdienste) und über Schrittfolgen (Anleitungsdienste) Kernelemente des Prozessmetamodells darstellen. Zusätzlich werden im Prozessmodell elementare Dienste als atomare Prozessschritte referenziert. Somit muss das Prozessmetamodell alle drei Diensttypen geeignet repräsentieren können. Das Werkzeugmetamodell muss geeignete Konzepte für die Modellierung der Werkzeugfunktionalitäten bereitstellen. Dies bezieht sich zum einen auf die elementaren Dienste, aber auch auf die Modellierung der Interaktionsmöglichkeiten bei der Operationalisierung von Beratungsdiensten.

5.3 Modellierung von Prozessfragmenten

5.3.1 Ziele und Anforderungen

Die im vorangegangenen Abschnitt vorgenommene Betrachtung der Anforderungen an eine integrierte Prozess- und Werkzeugmodellierung haben ergeben, dass das Prozessmetamodell Konzepte zur *Spezifikation* von Beratungsdiensten (Benutzerauswahlen) und Anleitungsdiensten (Schrittfolgen) umfassen muss. Weiterhin erforderlich sind Konzepte zur *Referenzierung* elementarer Dienste, die von den Werkzeugen angeboten werden und im Werkzeugmodell genauer spezifiziert werden.

Im Kontext dieser Arbeit resultieren aus der Fokussierung auf entwicklerorientierte, kreative Entwurfsprozesse weitere Anforderungen an die Kernelemente einer geeigneten Prozessmodellierungssprache, zu denen im Wesentlichen die fragmentweise Darstellbarkeit von Prozessen, Kontextbasiertheit und Kompositionalität gehören:

- **Fragmentweise Darstellbarkeit von Prozessen**: Wie in Abschnitt 2.1.1 motiviert wurde, entziehen sich kreative Entwurfsprozesse in der Regel einer vollständigen und durchgängigen Präskription. Das bedeutet, dass es im Allgemeinen nicht möglich ist, für jede denkbare Situation innerhalb des Entwurfsprozesses festzulegen, welcher Arbeitsschritt als nächstes ausgeführt werden sollte. Daher gelingt die Prozessmodellierung nur fragmentweise für gutverstandene Abschnitte des Entwicklungsprozesses. Aus diesem Grund muss ein Prozessmodell als eine erweiterbare Sammlung unabhängiger oder nur lose gekoppelter Prozessfragmente organisiert werden können, ohne ein durchgängiges Prozessmodell zu erzwingen [LaBo97].
- **Kontextbasiertheit**: In Abschnitt 2.1.3 haben wir von einem Prozessunterstützungsansatz gefordert, dass die Unterstützungsleistung zum Zeitpunkt der Inanspruchnahme auf den aktuellen Prozesskontext zugeschnitten sein sollte (Kontextbezogenheit). Wegen der nur fragmentarischen Erfassung des Entwurfsprozesses im Prozessmetamodell kann jedoch nicht davon ausgegangen werden, dass in der Leitdomäne Informationen über den aktuellen Prozesskontext implizit aus dem Ausführungszustand eines globalen Prozessmodells abgeleitet werden kann. Dies bedeutet aber, dass für jedes modellierte Prozessfragment der *Kontext* beschrieben werden muss, in dem dieses durchgeführt werden kann.
- **Kompositionalität**: Gerade in kreativen Entwurfsdomänen ist die Akquirierung und modellmäßige Umsetzung von Prozesswissen ein Vorgang, der nicht mit der Erstellung eines initialen Prozessmodells abgeschlossen ist, sondern den Entwurfsprozess innerhalb eines Projekts und über Projektgrenzen hinweg ständig begleitet. Unterstützt durch Prozessverbesserungsinfrastrukturen (z.B. GQM [BaRo88; BaCR94]), wird hierbei ausgehend von einem anfangs möglicherweise nur rudimentären Verständnis elementarer Prozessschritte, d.h. der Basisfunktionalitäten der unterstützenden Werkzeug, Wissen über zunehmend komplexere Vorgehensweisen erworben, die sich durch Kombination bereits verstandener und im

Prozessmodell erfasster Prozessfragmente ergeben. Daher muss das Prozessmetamodell über geeignete Kompositionsmechanismen für die Komposition komplexerer Prozessfragmente aus einfacheren verfügen. Dabei sollten beratende, anleitende und automatisierende Prozessfragmente frei kombiniert werden können.

Wegen der Konzentration auf arbeitsplatzorientierte Entwurfsprozesse individueller Entwickler spielen eine Reihe von Modellierungsaspekten, die von auf das administrative Projektmanagement ausgerichteten Prozessmodellierungsansätzen betont werden, hier nur eine untergeordnete Rolle. Dazu gehören Modellierungskonzepte zur Organisationsmodellierung (Verantwortlichkeiten, Rollen etc.), zur Entwicklerkoordination (z.B. Freigabe und Weiterleitung von Dokumenten, Auftragserteilung etc.) und zur Ressourcenverwaltung (z.B. Kostenmodelle, Milestones, Deadlines etc.).

5.3.2 PRIME-PM: das NATURE-Prozessmetamodell

Zur Modellierung von Prozessfragmenten definieren wir in dieser Arbeit keine neue Prozessmodellierungssprache, sondern greifen auf ein Prozessmetamodell zurück, das im Rahmen des ESPRIT-Projekts NATURE [NATU96; JRSD99] entwickelt wurde [Gro*97; RoSM95; Pohl96; RoPB99]. Es stellt die oben geforderten Konzepte zur einheitlichen Modellierung von elementaren Diensten, Beratungsdiensten und Anleitungsdiensten bereit [PoWe97; Poh*99]. Das NATURE-Prozessmodell betont die *situative* Natur kreativer Entwurfsprozesse [Such87] und legt besonderes Gewicht auf die Modellierung des *Kontextes*, in dem sich der Entwickler in seiner Arbeitsumgebung befindet, auf seine *Entwurfsziele* und die dabei zu treffenden *Entscheidungen*. Gerade für kreative Entwurfsprozesse stellt das NATURE-Prozessmodell einen signifikanten Fortschritt gegenüber aktivitäts- oder produktzentrierten Prozessmodellierungssprachen dar, die sich vorwiegend auf Aussagen über Ausführungsreihenfolgen von Aktivitäten (das „Wie“) bzw. deren Resultate (das „Was“) beschränken und Entscheidungen und Situationen, in denen die Entscheidungen gefällt werden (das „Warum“), nur unzureichend repräsentieren [Gro*97].

Die Eignung des NATURE-Prozessmodells für die feingranulare Modellierung von Entwurfsmethodiken wurde in verschiedenen Anwendungsdomänen erfolgreich demonstriert, z.B. im Bereich des Requirements Engineering für die ER-Modellierung [PlRo95] oder die szenariobasierte Systemanalyse [RoSB98; HaPW98]. Weitere Anwendungserfahrungen des NATURE-Prozessmodells liegen für Modellierungsprozesse im Bereich der Verfahrenstechnik vor [WeBa99; Döm*96; Lohm98; EgKM00].

Im Folgenden erläutern wir kurz die wichtigsten Elemente des NATURE-Prozessmetamodells, das in Abb. 19 in Form eines UML-Klassendiagramms dargestellt ist.

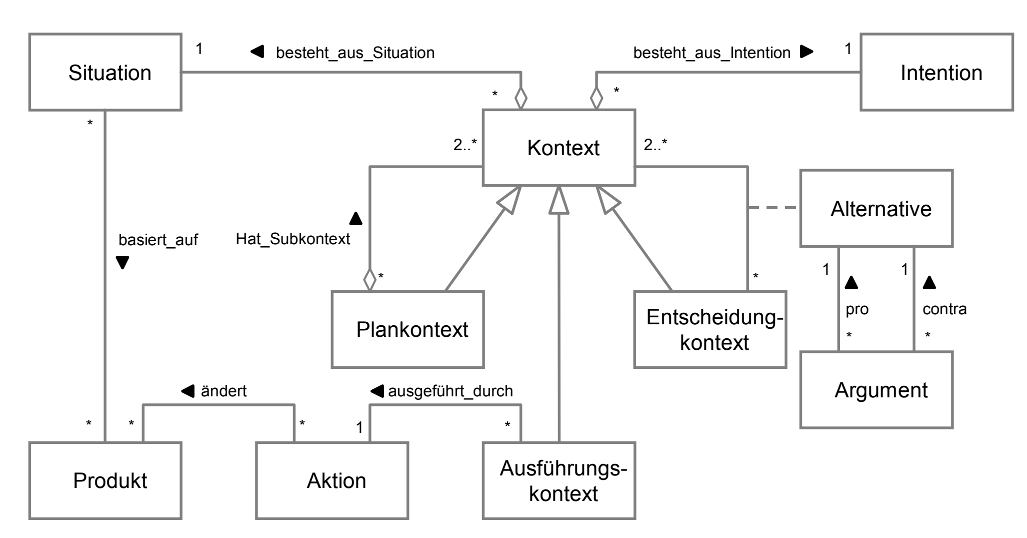


Abb. 19: PRIME-PM – das NATURE-Prozess-metamodell [Pohl96]

Prozessfragmente werden im NATURE-Prozessmetamodell mithilfe des zentralen Konzepts `Kontext` dargestellt. Ein als Kontext modelliertes Prozessfragment ist sowohl durch seine Aktivierungsbedingungen als auch durch die Art seiner Operationalisierung näher charakterisiert.

Eine wesentliche Grundidee des NATURE-Prozessmetamodells besteht in der expliziten Verknüpfung von Situationen, die der Entwickler zu einem Zeitpunkt vorfinden kann, mit den Intentionen, die er in diesen Situationen verfolgen kann. Die Beschreibung der Aktivierungsbedingungen eines Kontextes zerfällt somit in einen objektiven Anteil (die Situation) und einen subjektiven Anteil (die Intention). Ein Kontext gilt als aktiviert, wenn sowohl sein Situationsteil als auch sein Intentionsteil gültig sind.

Situation

Eine `Situation` beschreibt eine Konstellation des in Entwicklung befindlichen Produkts, über die es Sinn macht, eine Entscheidung zu treffen. Eine Situation ist somit als eine Abstraktion des aktuellen Produktzustands in einer Werkzeugumgebung zu verstehen. In Abb. 19 wird die Beziehung zwischen Situationen und Produkten durch eine einfache Assoziation (`basiert_auf`) repräsentiert. In Wirklichkeit ist diese Beziehung komplexer aufgebaut; sie definiert den strukturellen Aufbau einer Produktkonstellation aus atomaren und zusammengesetzten Produktteilen und ist eventuell durch zusätzliche Randbedingungen, die zum Vorliegen der Situation erfüllt sein müssen, näher beschrieben. Die formale Situationsspezifikation wird in Abschnitt 7.2.4 näher erläutert.

Produkt

Das Konzept `Produkt` steht stellvertretend für das zugrunde liegende Produktmodell und fasst alle während des Entwurfsprozesses anfallenden Informationen zusammen. Im engeren Sinne fallen hierunter zunächst die eigentlichen Entwurfsartefakte. In den von uns betrachteten frühen Phasen der Systementwicklung sind dies z.B. konzeptuelle Modelle wie ER-Diagramme, die wiederum aus einzelnen Modellierungselementen wie Entitätstypen und Beziehungstypen bestehen. Auf Produkten aufbauende Situationen können also auf unterschiedlichen Granularitätsebenen angesiedelt sein (komplexe Dokumente, individuelle Modellierungselemente) In einer erweiterten Sichtweise zählen wir auch so genannte *Supplementärprodukte* (Dokumentationen von Zielen, Entscheidungen, Begründungen etc., die zur Entstehung eines Produkts geführt haben), *Prozessbeobachtungsinformationen* (Informationen über abgelaufene Prozesse) sowie *Abhängigkeiten* zwischen den genannten Informationskategorien zum Produkt. Diese zusätzlichen Informationen dienen primär der Nachvollziehbarkeit der entstandenen Spezifikationen und

der dahinter liegenden Prozesse und bilden zusammen mit dem eigentlichen Produkt die so genannte *Prozessspur*. Eine detaillierte Taxonomie für die unterschiedlichen Kategorien von Produktinformationen ist in [Dömg99] zu finden. Die zur Detaillierung des Produktmodells benötigten Modellierungskonstrukte werden im Metamodell nicht weiter vorgegeben. Wir gegen davon aus, dass uns dafür die in der konzeptuellen Modellierung gängigen Konstrukte und Strukturierungshilfsmittel (Attribute, Assoziation, Aggregation, Spezialisierung etc.), wie sie etwa aus der UML oder O-Telos bekannt sind, zur Verfügung stehen.

Intention

Der subjektive Anteil eines Kontexts wird durch das Konzept der `Intention` dargestellt. Eine Intention spiegelt ein Ziel wider, das der Entwickler verfolgt. Genau wie Situationen können Intentionen auf unterschiedlichen Granularitätsebenen angesiedelt sein. Eine globale Intention könnte lauten, ein ER-Schema zu erstellen; eine lokale Intention könnte in dem Hinzufügen eines Attributs zu einem Entitätstypen bestehen.

Neben der Spezifikation der Situation und Intention zur Angabe seiner Aktivierungsbedingungen ist ein Kontext durch einen dritten Informationsblock, die Art seiner Operationalisierung, näher bestimmt. Das NATURE-Prozessmodell unterscheidet drei fundamentale Kategorien der Prozessunterstützung, die durch die Spezialisierungen Ausführungskontext, Entscheidungskontext und Plankontext dargestellt werden. Diese Spezialisierungen haben eine spezifische Semantik, die sich durch jeweils unterschiedliche Assoziationen zu anderen Konzepten des Prozessmodells ausdrücken.

Aktion

Auf der elementarsten Ebene kann der Arbeitsfortschritt innerhalb eines Prozessfragments direkt durch die Modifikation des in Entwicklung befindlichen Produkts erzielt werden. Eine solche Transformation ist Resultat einer deterministischen `Aktion`, die automatisiert oder zumindest strikt erzwungen ausgeführt wird. Kontexte, die auf diese Art operationalisiert werden können, werden im NATURE-Modell als `Ausführungskontexte` bezeichnet und sind über die Assoziation `ausgeführt_durch` mit der entsprechenden Aktion verknüpft. Die Ausführung einer Aktion `ändert` das Produkt und ruft so das Entstehen neuer Situationen hervor, die dann selbst wieder Gegenstand einer nachfolgenden Kontextaktivierung sein können. Beachtenswert ist, dass im Prozessmetamodell eine Aktion und der Kontext, in dem die Aktion eingesetzt werden kann, eigenständig modelliert werden. Dadurch lässt sich insbesondere ausdrücken, dass ein und dieselbe Aktion potenziell zur Umsetzung unterschiedlicher Kontexte mit variierenden Benutzerintentionen und Situationen dienen kann.

Ausführungskontext

Entscheidungskontext

`Entscheidungskontexte` modellieren Prozesszustände, in denen eine explizite Benutzerentscheidung über den weiteren Prozessablauf notwendig ist. Ein Entscheidungskontext schlägt wenigstens zwei `Alternativen` zur Umsetzung der Benutzerintention in der gegebenen Situation vor. Bei der Auswahl einer alternativen Vorgehensweise findet im Allgemeinen eine Verfeinerung der ursprünglichen Intention und/oder der Situation statt. Daher werden die Alternativen in rekursiver Weise selbst wieder als Kontexte modelliert. Im Prozessmetamodell werden die Vorgehensalternativen daher durch die Assoziationsklasse `Alternative` zwischen `Entscheidungskontext` und `Kontext` modelliert. Diese Klasse steht über die `pro`- und `kontra`-Assoziationen zusätzlich in Beziehung mit der Klasse `Argument`. Argumente können zur Beratung des Benutzers bei der Vorgehensauswahl zusätzlich angegeben werden. Im Gegensatz zu Automatisierungskontexten haben Ent-

Alternativen und Argumente

scheidungskontexte keine unmittelbaren Auswirkungen auf das in Entwicklung befindliche Produkt.

Plankontext

`Plankontexte` definieren eine aus mindestens zwei Teilschritten bestehende *Strategie* zur Erfüllung einer Intention in einer gegebenen Situation. Um einen Plankontext umzusetzen, müssen die einzelnen Teilschritte in einer vorgegebenen Reihenfolge abgearbeitet werden. Ähnlich wie bei Entscheidungskontexten werden die Teilschritte selbst wieder rekursiv als Kontexte modelliert (über die Assoziation `hat_Subkontext`). Anders als Entscheidungskontexte, die den Entwickler bei der Vorgehensauswahl beraten, unterstützen Plankontexte jedoch längerfristige Aktivitäten durch die Angabe einer Reihenfolgebeziehung zwischen Teilschritten. Mithilfe von Plankontexten lassen sich somit *Dekompositionsstrukturen* modellieren, während durch Entscheidungskontexte *Verfeinerungsstrukturen* dargestellt werden. Zur Darstellung des Kontrollflusses zwischen den Subkontexten eines Plankontexts wird in [RoSM95; Pohl96] das Prozessmodell um explizite Präzedenzgraphen zwischen Kontexten erweitert. In der in Abb. 19 dargestellten Version des Prozessmodells haben wir darauf jedoch verzichtet. Stattdessen werden wir in Kapitel 6 ein flexibleres Konzept vorstellen, das die Einbettung und interoperable Verwendung weitgehend beliebiger Kontrollflussformalismen im Rahmen des NATURE-Modells erlaubt.

Kontexthierarchien bilden Und-/Oder-Bäume

Aus der Tatsache, dass Entscheidungskontexte und Plankontexte selbst wieder rekursiv durch Kontexte beliebigen Typs verfeinert beziehungsweise dekomponiert werden können, resultiert eine freie Kombinierbarkeit von automatisierenden (Ausführungskontexten), beratenden (Entscheidungskontexten) und anleitenden Diensten (Plankontexten). Die durch die Schachtelung von Kontexten gebildeten Hierarchien weisen starke Parallelen zu den aus der Planungstheorie der Künstlichen Intelligenz bekannten Und-/Oder-Zielbäumen [MyCN92] auf. Die Assoziation zwischen einem Entscheidungskontext und seinen alternativen Kontexten entspricht einer Oder-Verknüpfung, während die Assoziation zwischen einem Plankontext und seinen Subkontexten mit einer Und-Verknüpfung korrespondiert. Ausführungskontexte bilden die Blätter eines aus einer Kontexthierarchie gebildeten Und-/Oder-Baums.

5.4 Modellierung von Werkzeugen

5.4.1 Ziele und Anforderungen

Wie in Abschnitt 5.2 motiviert wurde, existieren zwischen den Diensten, die im Prozessmodell definiert werden, und den Werkzeugen, die dem Entwickler in einer Entwurfsumgebungen zur Verfügung stehen, eine Reihe von Querbezügen. Um diese Querbezüge explizit zu machen und eine Zuordnung zwischen Prozessfragmenten und unterstützenden Werkzeugfunktionalitäten zu erreichen, müssen Werkzeuge zunächst auf der gleichen konzeptuellen Ebene modelliert werden wie Prozesse [Poh*99]. Aus den in Abschnitt 3.3 aufgestellten Anforderungen lassen sich drei wesentliche Aspekte ableiten, die in einem Werkzeugmodell erfasst werden müssen:

- **Produktmodell**: Im Prozessmodell (Abschnitt 5.3) werden bestimmte Konstellationen der in Entwicklung befindlichen oder bearbeiteten Pro-

dukte als situativer Bestandteil des Kontextbegriffs referenziert. Innerhalb der Durchführungsdomäne interagiert der Entwickler in erster Linie über seine Entwurfswerkzeuge mit dem zugrunde liegenden Produktmodell. Um die Datenintegration (siehe Anforderung A1, Abschnitt 3.3.2) mit dem Prozessmodell zu ermöglichen, müssen die prozessrelevanten Produktstrukturen, die in einem Werkzeug erzeugt, gelesen, modifiziert oder gelöscht werden können, im Werkzeugmodell erfasst werden. Da wir auf der Prozessmodellierungsebene inhaltsorientierte Aussagen über feingranulare Produktkonstellationen treffen, müssen die von einem Werkzeug bearbeiteten Produkte im Allgemeinen wesentlich detaillierter als auf der Ebene von Ein- und Ausgangsdokumenten modelliert werden.

- **Basisfunktionalitäten**: Elementare Dienste, die im Prozessmodell als atomare, automatisierbare Prozessschritte referenziert werden, werden durch Basisfunktionalitäten der Werkzeuge umgesetzt. Dadurch wird im Prozessmodell gleichsam ein Mindestbedarf an unterstützender Werkzeugfunktionalität definiert. Ein Abgleich mit den tatsächlich zur Verfügung stehenden Werkzeugdiensten gelingt nur, wenn die Basisdienste der Werkzeuge vollständig modelliert werden (siehe Anforderung A3: Integrierte Prozess- und Werkzeugbeschreibung, Abschnitt 3.3.4). Hierbei ist es wichtig, dass ein interaktives Werkzeug sowohl aus Benutzer- als auch aus Prozesssicht in der Regel nicht als ein monolithischer Basisdienst angesehen werden kann, sondern als eine Sammlung feingranularer, prozessrelevanter Dienste.
- **Interaktionsmöglichkeiten**: Gemäß der Forderung nach prozesssensitiven Benutzerschnittstellen in prozessintegrierten Werkzeugen (siehe Anforderungen A5 und A6, Abschnitte 3.3.6 und 3.3.7) resultieren die im Prozessmodell als Entscheidungskontext modellierten Beratungsdienste in einer Einschränkung der Interaktionsmöglichkeiten des Entwicklers in seinen Werkzeugen. Der Fokus des Entwicklers soll auf die aktuell relevanten Dienste und Produktteile gelenkt werden, und der Zugriff auf nicht erlaubte Dienste bzw. Produkte soll verhindert werden. Weiterhin sollen Prozessfragmente aus der Benutzerschnittstelle eines Werkzeugs aktiviert werden können. Die explizite Modellierung der Interaktionselemente eines Werkzeugs ist daher eine Grundvoraussetzung für eine dynamische und anpassbare Zuordnung zu den im Prozessmodell definierten Alternativen eines Auswahldienstes.

5.4.2 PRIME-TM: Das Werkzeugmetamodell

Im Folgenden stellen wir ein Werkzeugmetamodell vor, das die Modellierung der oben genannten Werkzeugaspekte erlaubt. Es wurde insbesondere mit dem Ziel einer einfachen Integrierbarkeit mit dem in Abschnitt 5.3 vorgestellten Prozessmetamodell entworfen (siehe auch Abschnitt 1.5).

Im Zentrum des Werkzeugmetamodells steht das Konzept der `Werkzeugkategorie` (siehe Abb. 20) Durch Instanziierung dieser Klasse können die in einer Entwurfsumgebung zur Verfügung stehenden Werkzeuge modelliert werden. Zum Modellierungszeitpunkt betrachten wir Werkzeugkategorien lediglich auf der Typebene, während zur Laufzeit durchaus mehrere konkrete Ausprägungen einer

bestimmten Werkzeugkategorie aktiv sein können, z.B. mehrere laufende ER-Editoren. Diese bezeichnen wir dann als Werkzeuginstanzen.

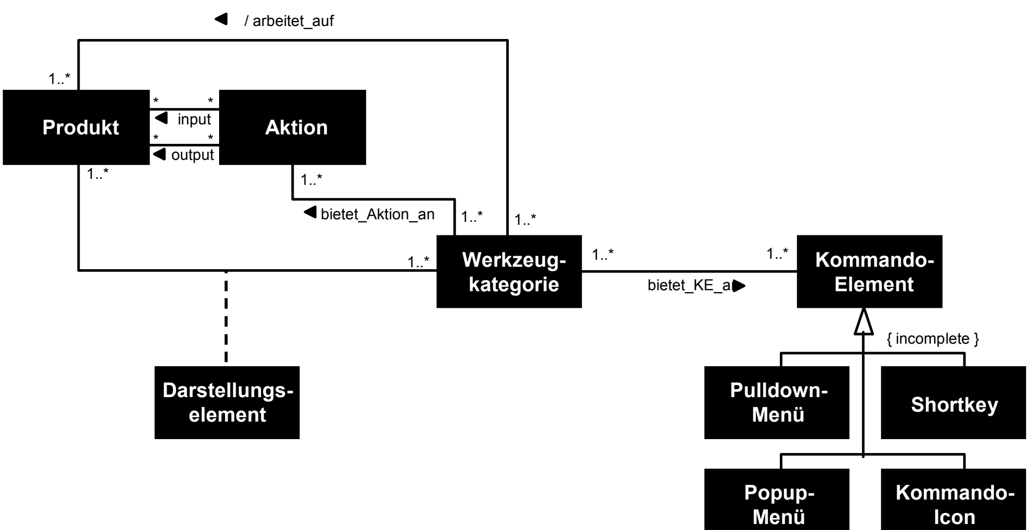


*Abb. 20: PRIME-TM – Das Werkzeug-metamodell [Poh*99]*

Mit Hilfe des Konzepts `Aktion` werden die von den Werkzeugen bereitgestellten elementaren Dienste modelliert. Die Verknüpfung von Aktionen zu den Werkzeugkategorien, die diese Aktionen bereitstellen, erfolgt über die Assoziation `bietet_Aktion_an` an.

Prozessneutrale Beschreibung von Werkzeug-Aktionen

Aktionen werden durch Angabe ihrer Eingangsprodukte, d.h. ihrer `input`-Parameter, und ihrer Ausgangsprodukte, d.h. ihrer `output`-Parameter, näher beschrieben. Im Gegensatz zum Prozessmetamodell werden Aktionen daher durch ihre unmittelbare Ein-/Ausgangsschnittstelle beschrieben, ohne sie in Bezug zu spezifischen Prozesskontexten zu setzen. Aktionen werden aus Werkzeugsicht also *prozessneutral* definiert.

Eine Werkzeugkategorie wird indirekt über die von ihr angebotenen Aktionen, die lesend oder schreibend auf bestimmte Produkte zugreifen, mit dem zugrunde liegenden Produktmodell verknüpft. Dies ist in Abb. 20 durch die abgeleitete Assoziation `/arbeitet_auf` angedeutet. Im korrespondierenden O-Telos-Modell lassen sich die einer Werkzeugkategorie zugeordneten Produkte mit Hilfe einer deduktiven Regel herleiten:

```
MetametaClass Werkzeugkategorie with
  attribute
    arbeitet_auf : Produkt
  rule
    r: $ forall p/Produkt a/Aktion
          (this bietet_aktion_an a) and
          ((a input p) or (a output p))  ==>
            (this arbeitet_auf p) $
end
```

Ähnlich dem Prozessmetamodell geben wir zunächst keine spezifischen Konzepte für die Detailmodellierung der den Werkzeugen zugrunde liegenden Produktstrukturen vor, sondern greifen auf die in konzeptuellen, objektorientierten Modellierungssprachen (in unserem konkreten Fall: UML und O-Telos) üblichen Ausdrucksmittel zurück (Attribute, Assoziation, Aggregation, Spezialisierung etc.). Das Konzept `Produkt` steht also stellvertretend für das einem Werkzeug zugrunde

liegende Produktmodell. Da wir das einer Werkzeugkategorie zugeordnete Produktmodell immer im Zusammenhang mit den darauf operierenden Aktionen betrachten, erhalten wir eine objektorientierte Sicht auf das Produktmodell.

Modellierungs der Interaktionsmöglichkeiten

Im Gegensatz zu den gängigen Werkzeugbeschreibungssprachen (siehe Abschnitt 3.3.4) bezieht unser Metamodell auch die Modellierung der Darstellungsmöglichkeiten für Produkte und der Interaktionsmöglichkeiten des Benutzers mit ein. Dies ist erforderlich, um später im Umgebungsmodell den Einfluss von Prozessdefinitionen und des aktuellen Prozesszustands auf die Benutzeroberfläche eines Werkzeugs modellieren zu können (siehe Anforderung A5 und A6 in Abschnitt 3.3.6 bzw. 3.3.7).

Darstellungsarten

Die grafische Repräsentation eines Produkts wird mithilfe der Assoziationsklasse `Darstellungselement` modelliert. Da diese Klasse nicht nur mit der Klasse `Produkt`, sondern auch mit der Klasse `Werkzeugkategorie` verknüpft ist, kann für jede Werkzeugkategorie, in der ein bestimmtes Produkt angezeigt werden kann, eine spezifische Darstellung gewählt werden. Die Klasse Darstellungselement selbst dient als Ankerpunkt für eine detaillierte Modellierung der Repräsentation eines Produkts durch eine bestimmte grafische Form (z.B. Rechteck, Kreis, Raute etc.). Weitere Darstellungsattribute (Farbe, Größe u.ä.), die für unterschiedliche Aktivierungszustände eines Produkts unterschiedlich gesetzt werden können, werden in der Klasse `Darstellungsart` zusammengefasst.

Kommandoelemente

Der Benutzer interagiert mit modernen interaktiven Benutzeroberflächen, indem er grafische Repräsentationen der Modellierungsprodukte auswählt und eine Funktion auf diesen Produkten durch ein entsprechendes `Kommandoelement` anstößt. Welche Kommandoelemente ein Werkzeug bereitstellt, wird durch die Assoziation `bietet_KE_an` definiert. Die Klasse `Kommandoelement` unterteilt sich in verschiedene Subklassen. In unserer aktuellen Implementierung (siehe Kapitel 7) unterstützen wir als in heutigen Benutzeroberflächen gängigste Kommandoelemente `Pulldown-Menüs`, `Popup-Menüs`, `Shortkeys` und `Kommando-Icons`.

5.5 Integration der Modelle

5.5.1 Ziel des Umgebungsmodells

Das Prozessmetamodell stellt Konzepte zur Modellierung von Prozessfragmenten als Ausführungs-, Entscheidungs- bzw. Plankontexte bereit. Mit den Konzepten des Werkzeugmetamodells werden Werkzeuge durch ihre Basisfunktionalitäten, das zugrunde liegende Produktmodell sowie ihre Interaktionselemente näher beschrieben. Ziel des Umgebungsmetamodells ist es, die im Prozessmodell definierten Kontexte auf die in einer Entwurfsumgebung zur Verfügung stehende Werkzeugfunktionalität abzubilden und dabei insbesondere den Einfluss von Entscheidungs- und Ausführungskontexten auf das Werkzeugverhalten festzulegen. Wie wir in Kapitel 7 sehen werden, werden die im Umgebungsmodell vorgenommenen Zuordnungen zwischen Prozessfragmenten und Werkzeugen von generischen Laufzeitkomponenten der Prozessmaschine, des Kommunikationsmechanismus und der Werkzeuge bei der Aktivierung, Ausführung und Koordination von Prozessfragmenten interpretiert.

5.5.2 PRIME-UM: Das Umgebungsmetamodell

Da das Werkzeugmetamodell in Hinblick auf eine Verknüpfung mit dem Prozessmodell entworfen wurde, ist die Integration von Prozess- und Werkzeugmetamodell auf recht einfache Weise möglich. Abb. 21 zeigt das Umgebungsmetamodell mit den jeweiligen Anteilen aus dem Prozessmetamodell (weiß unterlegte Konzepte) und aus dem Werkzeugmetamodell (schwarz unterlegte Konzepte). Konkret erfolgt die Integration über ein gemeinsames Metamodell für die Modellierung von Produkten und Aktionen (grau unterlegte Konzepte) und über zusätzliche Assoziationen zwischen Konzepten des Prozess- und Werkzeugmodells (gestrichelte Assoziationen).

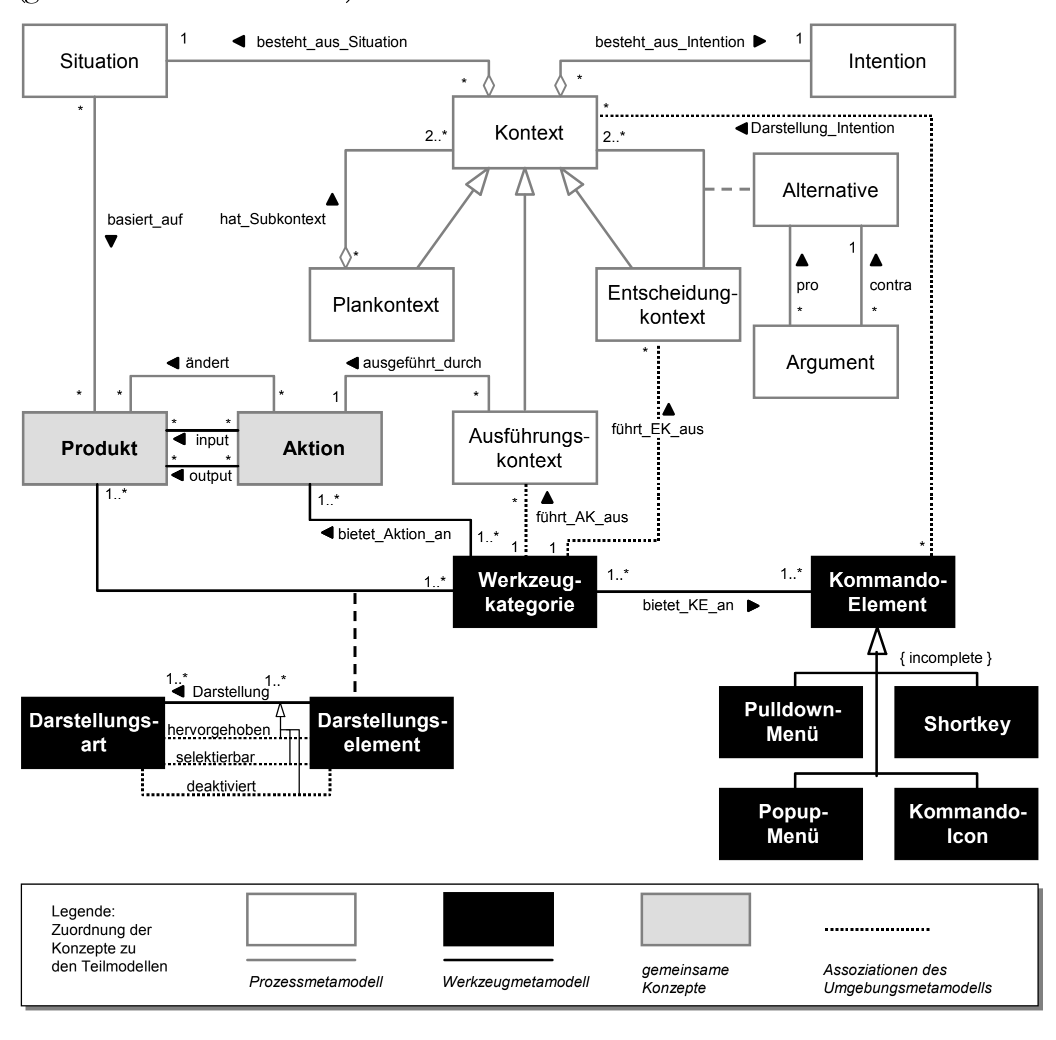


***Abb. 21**: PRIME-UM – Das Umgebungsmetamodell*

5.5.2.1 Abbildung von Ausführungskontexten auf Werkzeugkategorien

Um einen Ausführungskontext, der im Prozessmodell definiert wurde, operationalisieren zu können, muss dieser einer Werkzeugkategorie zugeordnet werden. Diese Verantwortlichkeit wird repräsentiert durch eine Instanz der Assoziation `führt_AK_aus`. Für eine korrekte Zuweisung von Ausführungskontexten zu Werkzeugkategorien lassen sich eine Reihe von Konsistenzbedingungen formulieren.

AK1: Existenz der Werkzeugunterstützung

Zunächst muss sichergestellt werden, dass die im Prozessmodell für die Operationalisierung eines Ausführungskontexts vorgesehene Aktion von einem der im Werkzeugmodell modellierten Werkzeuge angeboten wird. Sei *AK* ein im Prozessmodell definierter Ausführungskontext, *A* die mit *AK* assoziierte Aktion und $W_1, ..., W_n$ die im Werkzeugmodell definierten Werkzeugkategorien.

Bei der Zuweisung eines Ausführungskontexts zu einem Werkzeug können dann drei Fälle auftreten:

- **Automatische Zuweisung**: Es existiert genau eine Werkzeugkategorie $W_i \in \{W_1, ..., W_n\}$, die die Aktion *A* anbietet. In diesem Fall kann eine automatische Zuordnung zwischen *AK* und W_i erfolgen, da keine Wahlfreiheit existiert.
- **Wahl der Zuordnung**: Es existieren zwei oder mehr Werkzeugkategorien $\{W_{i1}, W_{i2}...,W_{ik}\} \subseteq \{W_1, ..., W_n\}$, die die Aktion *A* anbieten. In diesem Fall sollte der Methodeningenieur genau eine Werkzeugkategorie $W \in \{W_{i1}, W_{i2}...,W_{ik}\}$ auswählen und mit dem Ausführungskontext *AK* verknüpfen. Dies ist erforderlich, damit die Prozessmaschine, sobald der Ausführungskontext *AK* zur Ausführung ansteht, das zuständige Werkzeug aus dem Umgebungsmodell eindeutig bestimmen kann.
- **Mangel an Werkzeugunterstützung**: Wenn keine Werkzeugkategorie *W* die erforderliche Aktion *A* zur Verfügung stellt, ist dies ein Indiz dafür, dass der im Prozessmodell definierte Ausführungskontext von der aktuell im Werkzeugmodell erfassten Werkzeugfunktionalität der Entwurfsumgebung nicht unterstützt wird. Als Konsequenz muss eine entsprechende Werkzeugaktion zusätzlich realisiert und modelliert werden oder das Prozessmodell muss so angepasst werden, dass die fragliche Werkzeugaktion nicht länger erforderlich ist.

Welche der oben skizzierten Situationen vorliegt, kann durch folgende generische O-Telos-Anfrageklasse auf einfache Weise ermittelt werden:

```
GenericQueryClass WZ_Kategorie_für_Aktion isA Werkzeugkategorie
with
  parameter
    A: Aktion
  constraint
    AK1 : $ this bietet_Aktion_an ~A $
end
```

Je nach dem, ob die Anfrage eine, mehrere oder keine Werkzeugkategorie als Antwort liefert, liegt einer der obigen Fälle vor und der Methodeningenieur kann entsprechende Maßnahmen ergreifen. Ohne eine explizite Modellierung der Werkzeuge wäre ein solcher Abgleich zwischen der im Prozessmodell geforderten Werkzeugunterstützung und der tatsächlichen Werkzeugfunktionalität nicht möglich gewesen.

Dass eine Werkzeugkategorie *W* die vom Prozessmodell geforderte Aktion *A* für die Operationalisierung eines Ausführungskontexts *AK* bereitstellt, ist eine

notwendige, aber noch keine hinreichende Bedingung für eine konsistente Zuordnung zwischen *W* und *AK*. Zusätzlich muss sichergestellt werden, dass die `input`-/`output`-Relationen zwischen der Werkzeugaktion und den betroffenen Produkten von den entsprechenden Definitionen im Prozessmodell subsummiert werden.

AK2: Konsistenz bezüglich Input-Parametern

Im Prozessmodell werden die Produkte, auf die sich ein Ausführungskontext *AK* bezieht, durch die mit *AK* assoziierte Situation *S* festgelegt. Sei P_S die Menge der so mit *AK* indirekt assoziierten Produkten. Weiterhin sei P_A die Menge der Produkte, die im Werkzeugmodell über die Assoziation `input` mit der Aktion *A* verbunden ist. Dann muss $P_A \subseteq P_S$ gelten, d.h. die durch die Situation spezifizierte Produktkonstellation subsummiert die für die Durchführung der Aktion benötigten Eingangsprodukte. P_A kann dabei eine echte Teilmenge von P_S sein, da zur Bestimmung der Gültigkeit der Situation *S* durchaus eine größerer Produktausschnitt betrachtet werden kann, als zur eigentlichen Durchführung der Aktion *A* erforderlich ist. Dies lässt sich in O-Telos mithilfe der folgenden Integritätsbedingung `AK2` formalisieren, die zweckmäßigerweise der Attributkategorie `Werkzeugkategorie!führt_AK_aus` zugeordnet wird.

```
Attribute Werkzeugkategorie!führt_AK_aus with
  constraint
    AK2: $ forall p/Produkt ak/Ausführungskontext a/Aktion
           ( From(this, ak) and
             (ak ausgeführt_durch a) and (a input p) ==>
             ( exists s/Situation
               ( (ak besteht_aus_Situation s) and
                 (s basiert_auf p) ) )
        $
end
```

AK3: Konsistenz bezüglich Output-Parametern

Die Effekte einer Aktion *A* auf das in Bearbeitung befindliche Produkt werden im Prozessmodell durch die Assoziation `ändert` beschrieben. Aus Sicht der Werkzeugmodellierung werden die Ausgabeparameter einer (Werkzeug-)Aktion durch die Assoziation `output` beschrieben. Analog zur oben beschriebenen Integritätsbedingung `AK2` bezüglich der `input`-Parameter muss für die output-Parameter sichergestellt werden, dass die im Prozessmodell beschriebenen Änderungen am Produktmodell die im Werkzeugmodell definierten Ausgabeparameter subsummieren. Seien also P_C und P_O die Mengen der Produkte, die über die `ändert`- bzw. `output`-Assoziation mit der Aktion *A* verbunden sind. Dann muss $P_O \subseteq P_C$ gelten. Diese Forderung lässt sich durch die O-Telos-Integritätsbedingung `AK3` formalisieren, die wiederum der Assoziation `Werkzeugkategorie!führt_AK_aus` zugeordnet wird.

```
Attribute Werkzeugkategorie!führt_AK_aus with
  constraint
    AK3: $ forall p/Produkt ak/Ausführungskontext a/Aktion
           ( From(this, ak) and
             (ak ausgeführt_durch a) and (a output p) ==>
             (a ändert p ) )
         $
end
```

5.5.2.2 Abbildung von Entscheidungskontexten auf Interaktionselemente

Jeder im Prozessmodell definierte Entscheidungskontext *EK* wird über die Assoziation `führt_EK_aus` genau einer Werkzeugkategorie zugeordnet. Durch die im Prozessmodell spezifizierten Alternativen des Entscheidungskontexts wird somit ein Beratungsdienst für die Werkzeugkategorie *W* definiert. Die *Ausführung* von *EK* bedeutet für das Werkzeug *W*, dass es die Interaktionsmöglichkeiten des Benutzers auf die für *EK* definierten Alternativen anpassen muss und dem Benutzer die Auswahl einer Alternative ermöglichen muss. Gemäß der Struktur des Prozessmetamodells entsprechen die einzelnen Alternativen des Entscheidungskontexts *EK* wiederum Kontexten. Die alternativen Kontexte können beliebigen Typs (Ausführungs-, Entscheidungs- oder Plankontext) sein und selbst wieder einer anderen Werkzeugkategorie oder der Prozessmaschine (im Fall von Plankontexten) zugeordnet sein. Bei der Operationalisierung eines Entscheidungskontexts kommt es also nur darauf an, dass eine Werkzeugkategorie alle Alternativen in der Benutzeroberfläche zur Auswahl anbieten kann. Die *Aktivierung* eines alternativen Kontexts und seine *Ausführung* werden also unabhängig voneinander behandelt. Dadurch kann ein Werkzeug in die Lage versetzt werden, dem Benutzer die Funktionalitäten anderer Werkzeuge oder das Anstoßen von Prozessfragmenten anzubieten.

Die Informationen, wie die Alternativen eines Entscheidungskontexts darzustellen sind, werden aus Gründen der Anpassbarkeit nicht in der Implementierung der Werkzeuge hartkodiert, sondern im Umgebungsmodell explizit repräsentiert, so dass ein Werkzeug zur Laufzeit diese Definitionen interpretieren kann. Konkret benötigt das Werkzeug Informationen darüber, wie die aktuell geladenen Produktteile darzustellen sind und wie die Intentionen der alternativen Kontexte auf Kommandoelemente des Werkzeugs abgebildet werden.

Darstellung der Produkte

Produktseitig lassen sich bei der Ausführung eines Entscheidungskontexts die in einem Werkzeug angezeigten Produktinstanzen in drei verschiedene Gruppen einteilen:

- **Hervorgehoben**: die Produktinstanzen, die zur Situationsinstanz des aktuell aktivierten Entscheidungskontexts gehören, werden in der Darstellungsart `hervorgehoben` repräsentiert.

- **Selektierbar**: Instanzen von Produkten, die potenziell zu Situationen der Alternativen des aktuellen Entscheidungskontexts beitragen können, werden in der Darstellungsart selektierbar repräsentiert.
- **Deaktiviert**: Instanzen von Produkten, auf denen keine Situation der Alternativen des aktuellen Entscheidungskontexts basiert, werden in der Darstellungsart deaktiviert dargestellt.

Zur Modellierung der unterschiedlichen Darstellungsarten werden drei Spezialisierungen der Assoziation Darstellung zwischen den Klassen Darstellungselement und Darstellungsart eingeführt.

Die Menge der in der Darstellungsart hervorgehoben darzustellenden Produkte ergibt sich aus der Situationsinstanz eines auszuführenden Entscheidungskontexts. Die Mengen der selektierbaren bzw. deaktivierten Produktinstanzen lassen sich deklarativ durch folgende O-Telos-Anfrageklassen bestimmen.

```
GenericQueryClass SelektierbareProduktinstanzen isa Class with
  parameter
    ek : Entscheidungskontext
  constraint
    c  : $ exists p/Produkt s/Situation k/Kontext
           (~ek Alternative k) and
           (k besteht_aus_Situation s) and
           (s basiert_auf p) and
           (this in p) $
end

GenericQueryClass DeaktivierteProduktinstanzen isa Class with
  parameter
    ek: Entscheidungskontext
  constraint
    c  : $ not (this in SelektierbareProduktinstanzen[~ek/ek]) $
end
```

Die erste Anfrage verwendet eine so genannte Metaformel mit der Klassenvariablen p, da Produktinstanzen referenziert werden müssen, deren Klassen zum Modellierungszeitpunkt noch nicht bekannt sind. Generell haben die Anfragen jedoch den Nachteil, dass sie *alle* im Repository vorhandenen Produktinstanzen betrachten. Tatsächlich sind jedoch nur die aktuell im Werkzeug geladenen und angezeigten Produktinstanzen von Interesse. In der konkreten Implementierung (siehe Abschnitt 7.3) nutzen wir diese Einschränkung aus und können so die betroffenen Produktinstanzen effizient ermitteln.

Abbildung von Intentionen auf Kommandoelemente

Um die Aktivierung der Alternativen $C_1,...,C_n$ des Entscheidungskontexts *EK* in einer Werkzeugkategorie *W* zu ermöglichen, muss die Darstellung aller Intentionen von $C_1,...,C_n$ durch Kommandoelemente von *W* festgelegt werden. Da eine Intention (z.b. *lösche*) mit mehreren Kontexten (z.B. *lösche_Entität* und *lösche_Attribut*) assoziiert sein kann, ist eine kontextabhängige Zuordnung der Intention zu den Kommandoelementen erforderlich. Deshalb wird jeder alternative Kontext C_i (und nicht seine Intention!) über eine Assoziation vom Typ Darstellung_Intention mit einem oder mehreren Kommandoelementen verknüpft.

Für die korrekte Zuordnung von Entscheidungskontexten zu Werkzeugkategorien lassen sich wie bei der Assoziation von Ausführungskontexten zu Werkzeugkategorien eine Reihe von Konsistenzbedingungen formulieren. Wie oben bereits erläutert, muss eine Werkzeugkategorie *W* die alternativen Kontexte $C_1,...,C_n$ eines ihr zugeordneten Entscheidungskontexts *EK* zwar nicht selbst *ausführen* können. Es ist jedoch erforderlich, dass alle Alternativen potenziell in dem Werkzeug *aktiviert* werden können.

EK1: Aktivierbarkeit der Situationen

Dazu muss ein Werkzeug zum einen in der Lage sein, alle Produkte darstellen zu können, die potenziell zu den Situationen der alternativen Kontexte betragen. Diese Bedingung wird mithilfe eines O-Telos-Constraints formalisiert, welcher im Umgebungsmodell der Assoziation `führt_EK_aus` zwischen `Werkzeugkategorie` und `Entscheidungskontext` zugeordnet wird:

```
Attribute Werkzeugkategorie!führt_EK_aus with
  constraint
    EK1: $
      forall w / Werkzeugkategorie ek/Entscheidungskontext
             k/Kontext p/Produkt s/Situation
        ( From(w, this) and To(this,ek) and (ek alternative k)
          and (k besteht_aus_Situation s) and (s basiert_auf p))
          ==> (w Darstellungselement p) $
end
```

Mithilfe der Integritätsbedingung `EK1` wird also zugesichert, dass für jedes Produkt, das für die Situationsauswahl in einer Werkzeugkategorie relevant sein könnte, ein entsprechendes Darstellungselement existiert. In der Integritätsbedingung wird nicht explizit gefordert, dass für die Darstellungselemente auch die oben genannten Darstellungsarten `hervorgehoben`, `selektierbar` und `deaktiviert` existieren. Diese Zusatzbedingung ist bereits durch die im UML-Modell spezifizierten Kardinalitätsbedingungen (siehe Abb. 21) abgedeckt.

EK2: Aktivierbarkeit der Intentionen

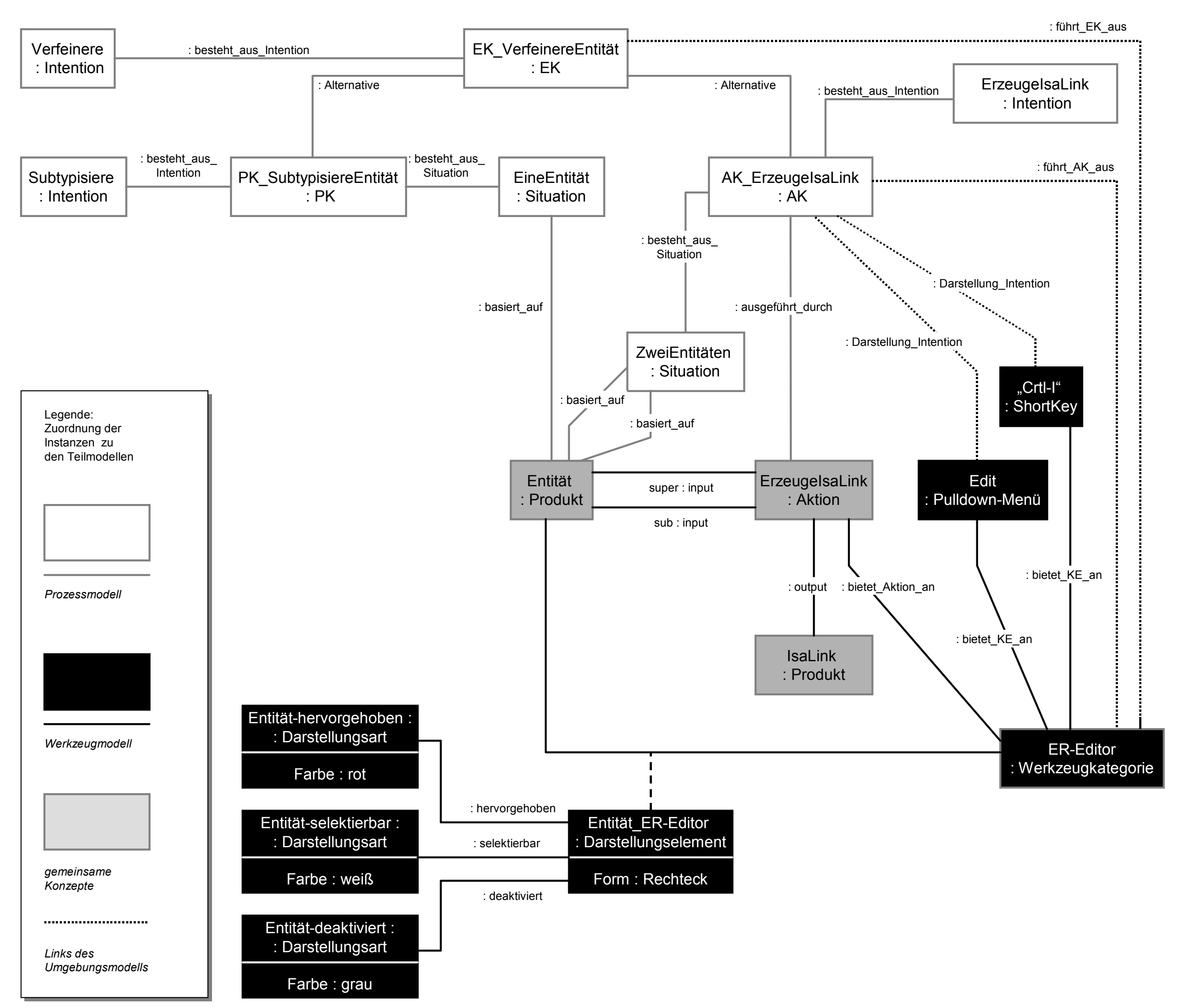
Um sicherzustellen, dass eine Werkzeugkategorie die Intentionen aller Alternativen eines ihm zugeordneten Entscheidungskontexts darstellen kann, muss jede Alternative über die Assoziation `Darstellung_Intention` mit der Werkzeugkategorie verknüpft sein.

```
Attribute Werkzeugkategorie!führt_EK_aus with
  constraint
    EK2 : $
      forall w / Werkzeugkategorie ek/Entscheidungskontext
             k/Kontext ke/Kommandoelement
        ( From(w, this) and To(this,ek) and (ek alternative k) )
           ==> ( exists ke/Kommandoelement
             (w bietet_KE_an ke) and (ke Darstellung_Intention k)
          ) $
end
```

5.6 Beispiel für ein Umgebungsmodell

Abb. 22 illustriert anhand eines kleinen Beispiels die Zuordnung der im Prozessmodell definierten Ausführungs- und Entscheidungskontexte zu den im Werkzeugmodell definierten Werkzeugressourcen. Der Übersichtlichkeit halber haben wir darauf verzichtet, die Klassenebene der Umgebungs*meta*modells explizit darzustellen. Stattdessen ist in der Darstellung bei den einzelnen Instanzen zusätzlich der Name der jeweiligen Klasse bzw. Assoziation angegeben.

***Abb. 22**: Beispiel für ein Umgebungsmodell (Ausschnitt)*



Der obere Teil von Abb. 22 stellt einen stark vereinfachten Ausschnitt aus einem Prozessmodell zur Entity-Relationship-Modellierung dar (vgl. [PlRo95]). Für den Entscheidungskontext `EK_VerfeinereEntität` werden zwei Alternativen definiert, nämlich der Ausführungskontext `AK_ErzeugeIsALink` und der Plankontext `PK_SubtypisiereEntität`. Mithilfe des Ausführungskontext `AK_ErzeugeIsALink` wird eine Entität durch eine andere verfeinert, indem eine Spezialisierungsbeziehung zwischen diesen beiden Entitäten erzeugt wird. Folglich beschreibt die Situation des Kontexts (`ZweiEntities`) eine Konstellation von zwei vom Benutzer ausgewählten Entitäten. Die andere Alternative von `EK_VerfeinereEntität`, der Plankontext `PC_SubtypisiereEntität`, beschreibt die Spezialisierung einer Entität

durch die Einführung einer neuen Sub-Entität. Dieser Kontext ist als Plankontext modelliert, da er mehrere Schritte umfasst (in der Abbildung nicht dargestellt: Erzeugen der Sub-Entität, Hinzufügen von Attributen, Einfügen einer Isa-Beziehung zur ursprünglichen Entität etc.). Seine Ausgangssituation basiert somit lediglich auf der zu spezialisierenden Entität (Situation EineEntität). Weiterhin sind im Prozessmodell die zu den Kontexten gehörenden Intentionen sowie die Aktion ErzeugeIsaLink mit den jeweiligen input- und output-Produkten dargestellt.

Der untere Teil von Abb. 22 zeigt einen Ausschnitt aus dem Werkzeugmodell. Hier ist die Werkzeugkategorie ER-Editor zusammen mit den unterstützten Kommandoelementen (dem Shortkey Crtl-I und dem Pulldown-Menü Edit) dargestellt. Weiterhin steht die Werkzeugkategorie ER-Editor mit gemeinsamen Konzepten aus dem Prozessmodell in Beziehung (mit der Aktion ErzeugeIsaLink und dem Produkt IsaLink).

Die Definitionen aus dem Prozess- und Werkzeugmodell werden mithilfe der Assoziationen aus dem Umgebungsmodell zueinander in Beziehung gesetzt:

- führt_AK_aus: entsprechend der Konsistenzbedingung AK1 (siehe Abschnitt 5.5.2.1) kann der ER-Editor automatisch dem Ausführungskontext AK_ErzeugeIsaLink zugeordnet werden, da er im aktuellen Umgebungmodell die einzige Werkzeugkategorie darstellt, welche die mit dem Ausführungskontext assoziierte Aktion ErzeugeIsaLink anbietet. Konsistenzbedingung AK2 wird erfüllt, da die input-Produkte der Aktion ErzeugeIsA-Link von der Situation des assoziierten Kontexts (ZweiEntitäten) subsummiert wird. In ähnlicher Weise genügt das Modell auch der Konsistenzbedingung AK3, da die Definition der output-Produkte (IsaLink) im Werkzeugmodell mit der ändert-Assoziation im Prozessmodell konform ist (in Abb. 22 aus Platzgründen nicht explizit dargestellt).
- führt_EK_aus: Über einen Link vom Typ führt_EK_aus wird der Entscheidungskontext EK_VerfeinereEntität als Beratungsdienst des ER-Editors definiert. Bei der Ausführung dieses Entscheidungskontexts muss der ER-Editor die beiden Alternativen AK_ErzeugeIsaLink und PK_Subtypisiere zur Auswahl anbieten. Dies erfordert, dass der ER-Editor die Intentionen (hier ErzeugeIsaLink und ErzeugeSubtyp) als Kommandoelemente und die von den Situationen betroffenen Produkte in verschiedenen Darstellungsarten darstellen kann.
  - Darstellung_Intention: Zwei Links vom Typ Darstellung_Intention ermöglichen die Aktivierung der Intention ErzugeIsaLink mithilfe des Shortkey Crtl-I und durch einen Eintrag im Pulldown-Menü Edit. In ähnlicher Weise erfolgt die Abbildung der Intention ErzeugeSubtyp auf die entsprechenden Kommandoelemente (in Abb. 22 nicht dargestellt). Somit wird die Konsistenzbedingung EK2 erfüllt.
  - Darstellungselement/Darstellungsart: Eine Instanz der Assoziationklasse Darstellungselement liefert Information darüber, wie der ER-Editor das Produkt Entität anzeigen soll, nämlich als Rechteck. Darüber hinaus geben Instanzen der Klasse Darstellungsart darüber Auskunft, wie eine Entität in den Zuständen hervorgehoben, selektierbar bzw. deaktiviert repräsentiert werden soll. Analog wird für das Produkt IsaLink das Darstellungselement (Pfeil) festgelegt und mit

entsprechenden Informationen über die unterschiedlichen Darstellungsarten angereichert (in Abb. 22 nicht dargestellt). Insgesamt wird dadurch die Konsistenzbedingung `EK1` bezüglich der Zuordnung zwischen der Werkzeugkategorie `ER-Editor` und dem Entscheidungskontext `EK_VerfeinereEntität` erfüllt, d.h. der `ER-Editor` ist in der Lage, alle Produkte, die zu Situationen der Alternativen von `EK_VerfeinereEntität` beitragen können, in den entsprechenden Darstellungsarten zu repräsentieren.

5.7 Fazit

Die Definition von Prozess- und Werkzeugmetamodell sowie deren Integration zu einem Umgebungsmodell bilden das konzeptuelle Fundament für die Prozessintegration von Werkzeugen. Der Methodeningenieur wird in der Zuordnung der im Prozessmodell definierten Prozessfragmente zu den entsprechenden Werkzeugfunktionalitäten unterstützt. Ähnlich dem Signaturvergleich von algebraischen Spezifikationen [ChHJ93] kann die korrekte Zuordnung von Ausführungs- und Entscheidungskontexten zu Werkzeugkategorien durch Überprüfen der als O-Telos-Constraints formalisierten Konsistenzbedingungen AK1 – AK3 sowie EK1 und EK2 gewährleistet werden.

Das Umgebungsmodell wird von den aktiven Hauptkomponenten einer prozessintegrierten Entwurfsumgebung (Prozessmaschine, Kommunikationsmechanismus und Werkzeuge) zur Laufzeit interpretiert. Die Prozessmaschine entnimmt dem Umgebungsmodell Informationen über Ablaufreihenfolgen bei der Abarbeitung von Plankontexten. Während der Plankontextinterpretation fordert die Prozessmaschine die Ausführung von Ausführungskontexten und Entscheidungskontexten von den Werkzeugen an. Der Kommunikationsmechanismus nutzt für die korrekte Verteilung der Kontextanforderungsnachrichten an die Werkzeuge die im Umgebungmodell definierte Zuordnung zwischen Ausführungs-/Entscheidungskontexten und Werkzeugkategorien. Die Werkzeuge rufen bei der Anforderung eines Ausführungskontexts die im Umgebungsmodell festgelegte Aktion auf bzw. passen bei der Anforderung eines Entscheidungskontexts ihre Benutzeroberfläche entsprechend an.

Insgesamt lässt sich der Beitrag des Modells zu den Kapitel 3 aufgestellten Anforderungen an eine Integration der Prozessdomänen wie folgt zusammenfassen:

- **A1 – Datenintegration**: Die Datenintegration zwischen den Prozessdomänen wird durch ein gemeinsames Produktmodell gewährleistet. Die Konsistenzbedingungen AK2 und AK3 stellen den Zusammenhang her zwischen den im Prozessmodell als Situationen definierten Produktkonstellationen und den input-/output-Parametern der von den Werkzeugen bereitgestellten Aktionen.
- **A2 – Prozessmediierte Werkzeuginteraktionen**: Das Wissen über werkzeugübergreifende *Abläufe* wird innerhalb des Prozessmodells durch das Konzept des Plankontexts repräsentiert. Spezifische *Arbeitsmodi* (Beratungsdienste) der Werkzeuge werden als Entscheidungskontexte modelliert. Wichtig ist, dass die in einem Entscheidungskontext auswählbaren Alternativen wiederum anderen Werkzeugen zugeordnet sein können. Da-

durch können aus einem Werkzeug heraus auch Prozessschritte aktiviert werden, die nicht von dem Werkzeug selbst operationalisiert werden. Mithilfe der Konzepte Plan- und Entscheidungskontext sind somit die wesentlichen prozessrelevanten Aspekte bei der Interaktion zwischen unterschiedlichen Werkzeugen explizit auf der Ebene der Prozessmodellierung repräsentiert und einer einfachen Adaptierbarkeit zugänglich.

- **A3 – Integrierte Prozess- und Werkzeugbeschreibung**: Prozessfragmente und Werkzeugaspekte werden auf der gleichen konzeptuellen Ebene repräsentiert und über gemeinsame Konzepte bzw. zusätzliche Assoziationen im Umgebungsmetamodell zueinander in Beziehung gesetzt. Gleichzeitig erlaubt der modulare Aufbau des Gesamtmodells aus einem Prozess- und einem Werkzeugmetamodells eine saubere Trennung zwischen prozessrelevanten und werkzeuginhärenten Aspekten. Das heißt, dass Prozesswissen in Form von Kontextdefinitionen zunächst unabhängig von einer Beschreibung der Werkzeuge definiert werden kann und umgekehrt Werkzeuge prozessneutral beschrieben werden können. Erst durch die Zuordnung zwischen Prozess- und Werkzeugmodell wird eine spezifische prozessintegrierte Umgebung konfiguriert. Die klare Separierung von Prozess- und Werkzeugaspekten stellt einen wesentlichen Fortschritt gegenüber einem früheren Modellierungsansatz dar, der ebenfalls auf dem NATURE-Prozessmodell beruhte und den Ausgangspunkt für diese Arbeit bildete [Weid95; Pohl96].
- **A4 – Feedback-Informationen für Synchronisation**: Die für die Synchronisation der Prozessdomänen erforderlichen prozesskonformen Rückmeldungen werden durch das Umgebungsmetamodell inhärent vorgegeben: bei Entscheidungskontexten durch die vordefinierten Alternativen, bei Ausführungskontexten durch die als `output` definierten Produkte der mit dem Ausführungskontext assoziierten Aktion. In Abschnitt 7.2 formalisieren wir ein Interaktionsprotokoll, das diese durch das Umgebungsmetamodell vorgegebene Struktur in den verwendeten Nachrichtentypen widerspiegelt.
- **A5 – Prozesssensitive Benutzeroberflächen**: Entscheidungskontexte modellieren Arbeitsmodi, die auf einen bestimmten Prozesskontext zugeschnitten sind. Bei der Ausführung eines Entscheidungskontexts findet ein Werkzeug im Umgebungsmodell alle erforderlichen Informationen darüber, welche Alternativen aktuell angeboten werden sollen und wie die Alternativen in der Benutzeroberfläche darzustellen sind (Darstellung der hervorgehobenen, selektierbaren und deaktivierten Produkte, Darstellung von Benutzerintentionen durch Kommandoelemente wie Menüeinträge, Kommandoicons und Shortkeys). Ein Werkzeug wird somit in die Lage versetzt, seine Benutzeroberfläche prozesssensitiv anzupassen.
- **A6 – Werkzeugunterstützter Aufruf von Prozessfragmenten**: Da Entscheidungskontexte insbesondere Plankontexte, die der Prozessmaschine zugeordnet sind, als Alternativen enthalten können, erlangt ein Werkzeug Wissen über die im aktuellen Prozesskontext verfügbaren Prozessfragmente und kann so den Aufruf extern definierter Prozessfragmente durch den Benutzer unterstützen.



Interoperabilität von Prozesssprachen

6.1 Motivation

Der im vorangegangenen Kapitel vorgestellte Prozess- und Werkzeugmodellierungsansatz repräsentiert werkzeugbezogene Prozessfragmente durch das zentrale Konzept des *Kontexts*, welches drei wesentliche Komponenten zusammenführt: einen *Intentionsteil*, einen *Situationsteil* und einem *Verhaltensteil*, wobei letzterer je nach Kontextkategorie (Ausführungskontext, Entscheidungskontext, Plankontext) unterschiedlich aufgebaut ist. Damit das Umgebungsmetamodell als *ausführbare* Prozessmodellierungssprache eingesetzt werden kann, muss es einem Laufzeitmechanismus alle zur Operationalisierung notwendigen Informationen bereitstellen. In der bisher beschriebenen Form lässt sich mit den Konzepten des Umgebungsmetamodell jedoch nur die Aktivierung von Intentionen sowie das Verhalten von Ausführungskontexten und Entscheidungskontexten vollständig spezifizieren. Zwei weitere Aspekte, die Definition von Situationen und die Festlegung von Abläufen zwischen den Subkontexten eines Plankontexts, sind im Umgebungsmetamodell noch *bewusst* offen gelassen (siehe Tab. 8).

***Tab. 8**:*
Informationen des Umgebungsmodells zur Operationalisierung

Kontextkomponente	**Informationen für Operationalisierung**		
Intention	PRIME-UM	✓	Aktivierung durch zugeordnete Kommandoelemente
Situation	*PRIME-UM*	×	*nicht vollständig spezifiziert*
	spez. Situationssprache erforderlich		- struktureller Aufbau - Situationsbedingung
Verhalten			
Ausführungskontext	PRIME-UM	✓	Ausführung durch zugeordnete Aktion und Werkzeugkategorie
Entscheidungskontext	PRIME-UM	✓	Angabe der Alternativen; Zuordnung zu Werkzeugkategorie; Abbildung der Alternativen auf Interaktionselemente
Plankontext	*PRIME-UM*	×	*nicht vollständig spezifiziert* *nur Angabe der Subkontexte*
	spez. Ablaufsprache erforderlich		Angabe des Kontroll- und Datenflusses zwischen Subkontexten

Spezifikation von Situationen

Wir haben bereits bei der Beschreibung des NATURE-Prozessmodells in Abschnitt 5.3.2 darauf hingewiesen, dass die durch die Assoziation `basiert_auf` repräsentierte Beziehung zwischen Situationen und Produkten in der Regel wesentlich komplexer aufgebaut ist; sie definiert den strukturellen Aufbau einer Produktkonstellation aus atomaren und zusammengesetzten Produktteilen und ist eventuell durch zusätzliche Randbedingungen, die zum Vorliegen der Situation

erfüllt sein müssen, näher beschrieben. Eine konkrete Sprache zur Situationsspezifikation hängt wesentlich vom zugrunde liegenden, domänenspezifischen Produktmodell und von den zur Verfügung stehenden Mechanismen zur Auswertung von Situationen ab. Die Detailspezifikation von Situationen als Sichten auf aktuelle Werkzeugzustände und entsprechende Auswertungskomponenten werden in Kapitel 7 genauer beschrieben.

Spezifikation von Abläufen in Plankontexten

Hauptgegenstand dieses Kapitels ist der zweite offene Aspekt des Prozessmetamodells, der die Ausführungsreihenfolge zwischen den Subkontexten eines Plankontexts betrifft. Über die Assoziation `hat_Subkontext` lässt sich bislang lediglich modellieren, welche Teilschritte zu einem Plankontext gehören, ohne jedoch eine präskriptive Abarbeitungsstrategie über der Menge der Subkontexte festlegen zu können.

Frühere Ansätze zur Anreicherung des NATURE-Prozessmodells

In früheren Ansätzen, die auf dem NATURE-Prozessmodell basieren, wurden zwei unterschiedliche Wege zur Anreicherung des Modells um Konzepte zur präskriptiven Ablaufspezifikation beschritten. In [RoSM95; Gro*97] wird das NATURE-Prozessmodell um *zusätzliche Konzepte* zur Kontrollflussspezifikation erweitert. Die Subkontexte eines Plankontexts werden über explizite, gegebenenfalls mit Bedingungen annotierte Präzedenzkanten in eine definierte Reihenfolge gebracht und bilden einen gerichteten Graphen. In [Pohl96] wird dagegen auf eine Erweiterung des NATURE-Prozessmodells um zusätzliche Kontrollflusskonstrukte und eine operationale Semantik verzichtet und stattdessen die Strategie verfolgt, eine *existierende* Sprache zur Ablaufspezifikation zu verwenden und die Konzepte des NATURE-Prozessmodells in dieser Sprache abzubilden. Konkret wurden hier Prozessfragmente als Methoden von Plankontext-Objekten in der Programmiersprache C++ formuliert und zur Prozessmaschine hinzugebunden. Außerdem wurde die Repräsentation des NATURE-Prozessmodells in dem Petrinetz-Dialekt SLANG untersucht und prototypisch in einer entsprechenden Prozessmaschine umgesetzt [Klam95].

Ziel hier: Interoperable Verwendung „beliebiger" Sprachen zur Ablaufspezifikation von Plankontexten!

In dieser Arbeit greifen wir den in [Pohl96; Poh*99] beschriebenen Ansatz der Einbettung des NATURE-Prozessmodells in existierende Ablaufformalismen auf, gehen aber noch einen Schritt weiter und untersuchen, wie sich potenziell *beliebige* Ablaufformalismen innerhalb des vom NATURE-Prozessmodell vorgegebenen Rahmens *interoperabel* verwenden lassen. Diese Zielsetzung ist durch die mittlerweile allgemein anerkannte Tatsache motiviert, dass die in den letzten 15 Jahren vorgeschlagenen Prozessmodellierungssprachen[17] jeweils spezifische Stärken und Schwächen aufweisen und kein einzelner Formalismus bzw. kein Sprachparadigma (z.B. prozedurale Sprachen oder Netzformalismen) als ideal anzusehen ist [ABGM93; GaLK98; HoVe97; GaJa96a]. Vielmehr stellen unterschiedliche Abläufe jeweils spezifische Ansprüche an einen Formalismus hinsichtlich seiner Ausdrucksstärke, d.h. seines Vorrats an Kontrollflusskonstrukten. Ist die Prozessmodellierungssprache wie in kommerziellen Workflowmanagementsystemen und den meisten akademischen Ansätzen a priori in Syntax und Semantik unveränderbar festgelegt, setzt dies dem Gestaltungsspielraum des Methodeningenieurs enge

[17] In diesem Kapitel konzentrieren wir uns auf den Kontrollfluss innerhalb von Plankontexten. Wenn von Prozessmodellierungssprachen die Rede ist, setzen wir dies daher mit einer Sprache zur *Ablaufspezifikation* zwischen Teilschritten gleich. Eventuell vorhandene andere Konzepte einer Prozessmodellierungssprache (Produktmodelle, Ressourcenmodelle, Organisationsmodelle etc.) sind somit bei dieser Diskussion ohne Belang.

Grenzen, denn jeder vordefinierte Sprachumfang bevorzugt eine bestimmte Anwendungsdomäne oder eine Klasse von Arbeitsvorgängen [DeHL96]. Eine Möglichkeit zur Flexibilisierung bieten Prozesssprachen, die dem Methodeningenieur die Einführung beliebiger neuer Kontrollflusskonstrukte erlauben [BMCJ98]. Erkauft wird diese Freiheit jedoch durch eine erhöhte Komplexität, da der Methodeningenieur nun neben der Sprachverwendung auch den Sprachentwurf bewältigen muss.

Der in dieser Arbeit verfolgte Ansatz geht einen Mittelweg und erreicht Flexibilität dadurch, dass der Methodeningenieur bei der Modellierung des Kontrollflusses eines kompositen Plankontexts aus einer erweiterbaren Palette *etablierter* Sprachen jeweils den Formalismus wählen kann, der für die Modellierung des aktuell betrachteten Ablaufs am geeignetsten ist und der seinen persönlichen Vorlieben am nächsten kommt. Dabei spielen neben der Frage nach den prinzipiellen Ausdrucksmöglichkeiten auch nichtfunktionale Anforderungen wie die Angemessenheit der Darstellung eine wichtige Rolle. Beispielsweise lassen sich mit Petrinetz-artigen Sprachen parallele Abläufe und Datenabhängigkeiten sehr anschaulich modellieren, während Konstrukte wie Schleifen nur umständlich repräsentiert werden können.

Aus der Verwendung unterschiedlicher Sprachen zur Kontrollflussspezifikation von Plankontexten und der potenziell beliebigen Schachtelung von Plankontexten ergibt sich das Problem der *Interoperabilität* heterogen spezifizierter Prozessfragmente. In diesem Kapitel beschäftigen wir uns mit den *modellierungsseitigen* Interoperabilitätsaspekten und entwickeln einen komponentenbasierten Integrationsansatz auf Basis des NATURE-Prozessmodells. Die ausführungsseitigen Konsequenzen zur Laufzeit werden in Kapitel 7 beleuchtet, wenn wir ein generisches Prozessmaschinen-Rahmenwerk für die Interpretation heterogen spezifizierter Prozessfragmente vorstellen.

Der Rest des Kapitels ist wie folgt gegliedert. In Abschnitt 6.2 geben wir einen kurzen Überblick über existierende Ansätze zur Interoperabilität prozessbasierter Systeme und motivieren einen komponentenorientierten Ansatz, in dem die Konzepte des NATURE-Prozessmetamodell als sprachneutrale Schnittstellenbeschreibung heterogener, in sich abgeschlossener Prozessfragmente fungieren. In Abschnitt 6.3 definieren wir ein auf dem NATURE-Prozessmodell basierendes Metamodell zur Schnittstellenbeschreibung. Gegenstand von Abschnitt 6.4 ist ein Bindungsmetamodell, das den Zusammenhang zwischen den Konzepten des Schnittstellenmetamodells und den Konstrukten eines konkreten Ablaufformalismus herstellt. Weiterhin skizzieren wir die Schritte einer auf diesem Modell basierenden Sprachintegrationsmethodik und illustrieren diese anhand der Einbindung der Formalismen SLANG und UML-Statecharts. Abschnitt 6.5 fasst die wesentlichen Ergebnisse des Kapitels zusammen.

6.2 Interoperabilität in prozessbasierten Systemen

Dieser Abschnitt gibt einen Literaturüberblick über existierende Interoperabilitätsansätze, mit denen der sprachliche Bruch zwischen unterschiedlichen Prozessmodellierungssprachen überbrückt werden kann. Wir beschäftigen uns zunächst mit existierenden Standards für die Interoperabilität prozessbasierter Systeme, gehen

dann auf einige Vorschläge für föderative Integrationsansätze ein und betrachten die Übertragung komponentenbasierter Spezifikationstechniken auf die Prozessmodellierung. Schließlich begründen wir, an welche der dargestellten grundsätzlichen Integrationsmöglichkeiten wir unser Konzept anlehnen.

6.2.1 Standards

Während im Bereich der Softwareprozessmodellierung noch keine Konsolidierung der unterschiedlichen Modellierungssprachen zu beobachten ist, entstand in der Workflowmanagement-Praxis mit der nunmehr breiten Verfügbarkeit von Werkzeugen für die Modellierung, Analyse, Simulation und Ausführung von Geschäftsprozessen der Wunsch, die Produkte unterschiedlicher Hersteller miteinander kombinieren zu können [Hay*00; Schu99]. Als wichtigste Standardisierungsaktivitäten für die Interoperabilität prozessbasierter Systeme sind das Workflow Reference Model der Workflow Management Coalition [Holl95] und der OMG Workflow Management Facility Standard (auch als *jointFlow*-Spezifikation bekannt) [OMG#98a] zu nennen.

6.2.1.1 WfMC-Referenzmodell

Das Workflow Management Referenzmodell der WfMC definiert die Interoperabilität zwischen unterschiedlichen Workflowmanagementsystemen beziehungsweise zwischen deren Teilsystemen. Der Grad der Interoperabilität zwischen den Prozessmaschinen unterschiedlicher Workflowmanagementsysteme wird in insgesamt acht Interoperabilitätsstufen eingeteilt:

- Stufe 1: Auf dieser Stufe existiert *keine Interoperabilität* zwischen den betrachteten Workflowmanagementsystemen.
- Stufe 2: Zwei unabhängige Prozessmaschinen, die die gleiche Hard- und Softwareinfrastruktur für die Ausführung eines unternehmensweiten Geschäftsprozesses nutzen, können *koexistieren*, ohne sich gegenseitig negativ zu beeinflussen. Zwischen den Prozessmaschinen erfolgt jedoch bei der Ausführung eines Teilprozesses keine Interaktion, so dass die Synchronisation der Teilprozesse ein manuelles Eingreifen erfordert.
- Stufe 3: Teilaspekte der Prozessdefinition können von zwei Prozessmaschinen geteilt werden, indem jeweils ein *spezifischer Überbrückungsmechanismus* den Austausch von z.B. Produktdaten ermöglicht.
- Stufe 4: *Eingeschränkte Kernfunktionalitäten* der Prozessmaschinen können über *offene Schnittstellen* wechselseitig in Anspruch genommen werden.
- Stufe 5: Die *volle Funktionalität* eines Workflowmanagementsystems steht über *offene Schnittstellen* zur Verfügung, so dass die Ausführung von Prozessfragmenten auf anderen Prozessmaschinen angestoßen werden kann. Diese Schnittstelle ist von der WfMC durch den Workflow Definition Interface Standard definiert [WfMC96].
- Stufe 6: Es existiert eine *gemeinsame Repräsentation* für Prozessmodelle. Die Prozessmodelle können durch ein gemeinsames *Austauschformat* zwischen den Prozessmaschinen transferiert werden, so dass die Wahl der Prozessmaschine für die Ausführung nicht relevant ist. Dies ermöglicht eine Integ-

ration der zuvor getrennten Teilprozesse der jeweiligen Prozessmaschinen in einem zentral administrierten Gesamtprozess.

- Stufe 7: Der Aufruf von APIs (inklusive Übertragung von Prozessmodellen und Arbeitseinheiten, Recovery etc.) wird durch ein *standardisiertes Protokoll* geregelt.
- Stufe 8: Die Komponenten der zu integrierenden Workflowmanagementsysteme weisen eine *einheitliche Benutzungsschnittstelle* auf.

Bis einschließlich Stufe 5 (Zugriff auf die volle Funktionalität über offene Schnittstelle) orientiert sich die Interoperabilität eher an den Funktionsbereichen eines prozessbasierten Systems (Leitdomäne) und weniger an den Konzepten der Prozessmodellierung selbst (Modellierungsdomäne). Neben der Schnittstelle zwischen den Prozessmaschinen unterschiedlicher Workflowmanagementsysteme unterscheidet das WfMC-Referenzmodell hierzu weitere Teilsysteme eines prozessbasierten Systems: Prozessmodellierungswerkzeuge, Administrationswerkzeuge und Klientenapplikationen (siehe Abb. 23). Das Referenzmodell legt Schnittstellen zwischen den einzelnen Teilsystemen fest und definiert dafür jeweils einen Standard. Die (seit der Version 2 zusammengelegten) Schnittstellen 2 und 3 für die Interaktion mit externen Applikation haben wir bereits in Abschnitt 3.3.4.2 (Seite 91ff) kennen gelernt.

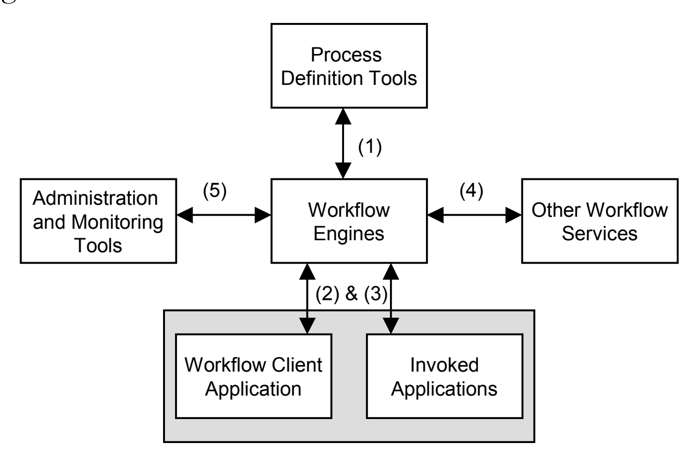


***Abb. 23**: Das Workflow-Referenzmodell der WfMC [Holl95]*

6.2.1.2 Austauschformate

WPDL

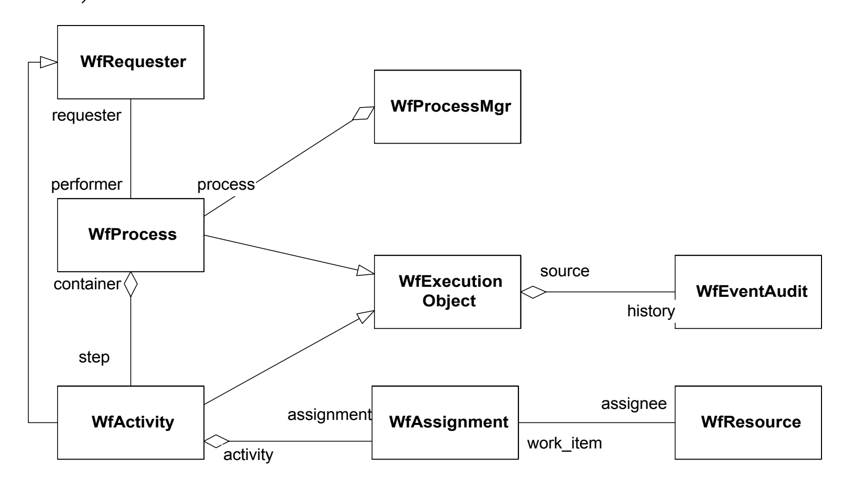
Die in unserem Kontext relevante Integration von Prozessfragmenten auf *Modellierungsebene* wird im WfMC-Referenzmodell erst auf der Interoperabilitätsstufe 6 betrachtet. Im Rahmen des Workflow Process Definition Interface WAPI-1 (siehe Abb. 23) hat die WfMC dazu die Sprache *WPDL* definiert, die als neutrales Zwischenformat für den Im- und Export von Prozessfragmenten zwischen den Prozessmodellierungswerkzeugen unterschiedlicher Workflowmanagementsysteme dient. Dazu müssen auf beiden Seiten entsprechende Konverter vorhanden sein, die die in den jeweiligen proprietären Sprachen formulierten Prozessmodelle in eine WPDL-Darstellung überführen. Der Grundgedanke von WPDL sowie weiterer Austauschformate (siehe unten) besteht darin, die Konzepte der Prozesssprachen in einer *einzigen Zwischensprache* zu vereinigen und zu homogenisieren, die dann als Integrationspunkt zwischen den einzelnen prozessbasierten Systemen genutzt

werden kann. Die *Prozessdefinition* in WPDL beschreibt die Elemente eines Workflows und umfasst die Deklaration der beteiligten *Aktivitäten*, *Transitionen*, *Applikationen* und *Prozessdaten*. Der Kontrollfluss zwischen Aktivitäten wird mit Transitionen spezifiziert, die eine sequentielle oder parallele Ausführungsreihenfolge festlegen. Für den Export eines Prozessfragmentes werden die beteiligten Aktivitäten, Applikationen und Prozessdaten im Format der WPDL deklariert. Ebenso wird das Kontrollmodell der jeweiligen Prozesssprache auf bedingte Transitionskanten der WPDL abgebildet, um so die Ausführungsreihenfolge unter den exportierten Aktivitäten festzulegen.

jointFlow

Die WAPI-Standards und die WPDL betrachten zwei duale Aspekte bei der Interoperabilität von Workflowmanagementsystemen. Während die WAPI-Standards Laufzeitschnittstellen für die Integration in der Leitdomäne definieren, liefert die WPDL eine sprachliche Integrationsbasis in der Modellierungsdomäne. Eine Vereinheitlichung der Modellierungs- und Ausführungssicht strebt das bei der OMG standardisierte jointFlow-Objektmodell [OMG#98a] an. Es liefert ein objektorientiertes Rahmenwerk für die Ausführung, Überwachung und Interoperabilität von Geschäftsprozessen und bildet Prozesskomponenten zusammen mit ihrer Ausführungsfunktionalität durch entsprechende Klassendefinitionen ab. Das jointFlow-Rahmenwerk besteht aus einer Spezifikation der Schnittstellen und Beziehungen der an der Operationalisierung eines Prozesses beteiligten Klassen (siehe Abb. 24):

Abb. 24: Das jointFlow-Objektmodell [OMG#98a]



- **WfRequester**: *WfRequester* sind Entitäten, die die Ausführung eines Prozesses anfordern. Dazu gehören sowohl Prozessmaschinen als auch externe Applikationen. Die Zustandsänderungen der Prozessausführung, wie z.B. die Termination eines Prozesses, können an den Requester weitergeleitet werden.
- **WfProcess**: Prozesstypen werden durch die Entität *WfProcess* repräsentiert, die eine Schnittstelle für die Prozesssteuerung anbietet, wie z.B. das Starten, Beenden oder Suspendieren. Ein WfProcess setzt sich aus mehreren WfActivities zusammen, die die Subschritte des Geschäftsprozesses darstellen. Instanzen eines Geschäftsprozesstyps werden durch Objekte mit der Schnittstelle WfProcess repräsentiert

- **WfProcessManager**: Ein *WfProcessManager* ist eine Fabrik für die Erzeugung einer Instanz eines Prozesstyps und hält Metainformationen über die verschiedenen Geschäftsprozesstypen.
- **WfActivity**: Eine *WfActivity* ist ein atomarer Geschäftsprozess, der von einem Agenten ausgeführt wird. Die Aktivität kann Teil eines übergeordneten Geschäftsprozesses sein, der einen weiteren Geschäftsprozess ausführt. In diesem Fall ist die Aktivität ein WfRequester eines weiteren Geschäftsprozesses, die die Ausführung eines Prozesses anfordert.
- **WfExecutionObject**: Ein *WfExecutionObject* vereinigt die Konzepte WfProcess und WfActivity.
- **WfResource**: *WfResourcen* stellen die für die Ausführung einer Aktivität notwendigen Ressourcen, wie z.B. Personen oder Rechnerkapazitäten dar.
- **WfAssignment**: Die Zuordnung von Ressourcen zu Aktivitäten wird durch *WfAssignments* repräsentiert.
- **WfEventAudit**: Ein *WfEventAudit* repräsentiert die Ereignisse, die während der Ausführung eines Prozesses aufgetreten sind. Durch die Schnittstelle kann innerhalb der Historie der Prozessausführung navigiert werden, so dass zeitliche Informationen wie z.B. die Termination eines Geschäftsprozesses auch nach der Ausführung weiterhin verfügbar sind.

Das jointFlow-Rahmenwerk steht in engem Zusammenhang mit den Standardisierungsbemühungen der WfMC. Die von der WfMC identifizierte Schnittstellenfunktionalität für die Interoperabilität zweier Prozessmaschinen wird vom jointFlow-Rahmenwerk in Form von Methoden, die den einzelnen Klassen zugeordnet sind, und Aufrufprotokollen übernommen. Im Gegensatz zur WfMC, die sich eher an den Schnittstellen der Teilsysteme eines Workflowmanagementsystems orientiert, liegt der Schwerpunkt von jointFlow auf den Schnittstellen der Entitäten eines Prozessmodells.

Weitere Austauschformate

Neben WPDL und jointFlow existieren noch weitere Austauschformate wie z.B. die aus der Fertigungsindustrie stammende Sprache *PSL* [ScKR96] und die im akademischen Umfeld entstandene Sprache *PIF* [LeYo94]. Analog zur WPDL werden die Konzepte der Prozessmodellierung bei PIF in einem Metamodell vereinheitlicht, welches aus einer hierarchisch angeordneten Anzahl von Klassen besteht. Die Klassenhierarchie wird ausgehend von der Wurzel *Entity* in die Klassen *Activity*, *Ressource* und *Relation* spezialisiert und ist in weitere durch PIF vorgegebene Subklassen aufgeteilt. Kontrollflusskonstrukte werden in PIF als spezielle Aktivitäten dargestellt, die je nach Bedarf verfeinert werden können. Somit sind, im Gegensatz zu den Transitionskanten der WPDL, die Kontrollflusskonstrukte in PIF auch erweiterbar. Die Vor- und Nachbedingungen von Aktivitäten werden in der Sprache *KIF* [GeFi92] formuliert.

6.2.2 Föderierte Interoperabilitätsansätze

6.2.2.1 ProcessWall-Ansatz

Interoperabilität über zentralen Zustandsserver

Prozessaustauschformate erzielen Interoperabilität zwischen Prozesssprachen durch Transformation über eine Zwischensprache, d.h. der Integrationspunkt liegt in der Modellierungsdomäne. In [Heim92] wird mit der *ProcessWall*-Umgebung ein Interoperabilitätsansatz vorgeschlagen, bei dem der Integrationspunkt von der Modellierungs- in die Leitdomäne verschoben wird. Die wesentliche Idee hinter diesem Ansatz besteht darin, dass der Prozesszustand mehrerer zusammenarbeitender Prozessmaschinen von dem Formalismus, d.h. der Ablaufmodellierungssprache, getrennt werden kann, der ihn modifiziert. Dazu verwaltet ein zentraler Zustandsserver zur Laufzeit den globalen Zustand der Prozessausführung. Der Prozesszustand wird als gerichteter, azyklischer Graph repräsentiert, in dem Aktivitäten durch Konten dargestellt werden, die durch Präzedenz- oder Subaktivitätskanten miteinander verbunden sind. Im Gegensatz zur statischen Repräsentation eines Prozessfragments spiegelt der Graph jedoch den Ausführungszustand eines virtuellen, über mehrere Prozessmaschinen verteilten globalen Prozessmodells zur Laufzeit wider.

Die Modifikation des den Prozesszustand repräsentierenden Graphen erfolgt durch unterschiedliche prozessbasierte Klienten, die mit dem Zustandsserver in Verbindung stehen. Diese Klienten melden über einen Nachrichtenmechanismus prozessrelevante Zustandsänderungen und können gegebenenfalls selbst auf Änderungen reagieren. Die Klienten des Zustandsservers werden unterschieden in Prozesskonstruktorklienten, Werkzeugklienten und Prozessbeschränkungsklienten. Die Prozesskonstruktorklienten, d.h. die Prozessmaschinen, expandieren den Zustandsgraphen, indem sie dynamisch Aktivitäten hinzufügen. Die Änderung des Graphen wird an interessierte Klienten, z.B. Werkzeuge, propagiert, so dass diese daraufhin die hinzugefügte Aktivität wahrnehmen und ausführen. Die erfolgreiche Ausführung einer Aktivität wird wiederum dem Zustandsserver mitgeteilt, der dann die Aktivität aus dem Graphen entfernt. Die Prozessbeschränkungsklienten überwachen den Prozesszustand und greifen gegebenenfalls ein, wenn dieser nicht mehr in einem konsistenten Zustand ist.

6.2.2.2 APEL

Die *APEL*-Architektur für föderierte prozessbasierte Systeme [DaEA98; EsBa98; EsCB98; EsDa96] basiert u.a. auf den mit ProcessWall gemachten Erfahrung, verfolgt aber eine etwas andere Zielrichtung. Im Gegensatz zu den zuvor erläuterten Modellen, die auf die Kompatibilität des Dienstangebots unterschiedlicher prozessbasierter Unterstützungsdienste ausgerichtet ist, sollen in der APEL-Architektur mehrere existierende PZEUen mit einem komplementären Dienstangebot zu einer nach außen hin einheitlich erscheinenden PZEU zusammengeführt werden. Diese virtuelle PZEU bietet dann Dienste, z.B. Arbeitsplatzverwaltungs-, Benachrichtigungs- und Prozessüberwachungsdienste, an, die nicht Teil einer jeden PZEU sind. Die einzelnen Dienste werden durch ein explizites Interoperabilitätsmodell zu einem homogenen Dienstangebot zusammengeführt, so dass sich die Dienstausführung der virtuellen PZEU auf diese Dienste stützen kann.

6.2.3 Prozesskomponenten

Die Abhängigkeit der in Abschnitt 6.2.1 vorgestellten Prozessaustauschformate von Transformationsbeziehungen zwischen Prozesssprachen kann durch *Komponentenansätze* überwunden werden, die über *Schnittstellen* die interne Ablauflogik eines Prozessfragments und den dafür verwendeten Implementierungsformalismus kapseln. Bei der Verwendung eines Prozessfragments als Subkomponente eines anderen Fragments muss nur noch die Schnittstelle des einzubettenden Fragments in die Sprache des übergeordneten Prozessfragments transformiert und entsprechende Bindungen definiert werden, so dass – eine entsprechende Laufzeitumgebung vorausgesetzt – prinzipiell auch stark heterogene Formalismen interoperieren können.

Wichtiges Unterscheidungsmerkmal von Komponentenmodellen allgemein sind der zugrunde liegende Komponentenbegriff, d.h. welche Prozessentitäten als Komponenten verstanden werden und wie diese strukturiert sind, sowie die Art der Komposition und Interaktion der Komponenten. Die Art des Kompositions- und Interaktionsmechanismus legt dann fest, wie die einzelnen Komponenten gekoppelt werden und wie die Dynamik zwischen den Komponenten spezifiziert wird.

6.2.3.1 Open Process Components

Das *Open Process Components*-Rahmenwerk (OPC) [GaLK98] ist ein objektorientiertes Interoperabilitätsmodell für Prozesskomponenten, welches in drei Abstraktionsschichten gegliedert ist. Die oberste *Modellschicht* basiert auf dem *PCIS LCPS*-Metamodell [Dern94] und identifiziert elementare Prozessentitäten als Klassen. Die Modellschicht legt mögliche Beziehungen zwischen Prozessentitäten fest: Prozesse können aus atomaren Aktivitäten zusammengesetzt werden oder stehen in Beziehung zu einem Subprozess. Rollen werden Aktivitäten zugewiesen und können durch Agenten ausgeführt werden. Produkte setzen sich aus Teilen zusammen und sind Eingabe oder Ergebnis einer Aktivität. Die Modellschichtklassen stellen Schnittstellen bereit, mit denen die Beziehungen unter den Prozessentitäten dynamisch hergestellt werden. Im Falle der Prozessklasse werden weiterhin elementare Dienste der Prozessausführung, wie z.B. das Starten, Beenden oder Terminieren eines Prozesses, angeboten.

Auf der *Repräsentationsschicht* wird die Prozessklasse der Modellschicht in die einzelnen Prozesssprachen-Paradigmen spezialisiert und um Schnittstellendienste erweitert, die für den jeweiligen Formalismus erforderlich sind. Die Schnittstelle eines Ereignisprozesses als Spezialisierung der Prozessklasse ist z.B. um die Methode *receiveEvent* ergänzt, so dass Ereignisse entgegengenommen werden können.

Die spezialisierten Prozessklassen der Repräsentationsschicht werden dann weiter auf der *Komponentenschicht* in die eigentlichen Prozesskomponenten verfeinert, welche die anwendungsspezifische Ablauflogik in sich tragen. Die Klassen der Komponentenschicht stellen die Prozesstypen wie z.B. *Design* oder *CodeTest* dar, deren Instanzen mit den Instanzen anderer OPC-Klassen, wie z.B. Agenten oder Produkten, in Beziehung stehen. Durch die geerbte Schnittstellenfunktionalität der Modellschicht können die Komponenten auf der Komponentenschicht kommunizieren, wobei die Sprache des Prozessfragments, welches das Kompo-

nentenverhalten spezifiziert, auf der Repräsentationsschicht durch die Schnittstellen verborgen wird. Ein OPC-Prozessmodell besteht somit aus einer Anzahl von Komponenten, die in durch die Modellschicht beschränkten Beziehungen stehen und deren Interaktion über definierte Schnittstellen erfolgt.

6.2.3.2 Pynode

Prozesskomponenten des Pynode–Projektes [AvBC96; AvBC96a] bestehen aus einer Verbindungsschnittstelle und einem Prozessfragment, welches die Schnittstelle in einer wahlfreien Prozesssprache implementiert. Die Verbindungsschnittstelle besteht zum einen aus Verbindungskanälen, die eine synchrone und asynchrone Datenübermittlung an die Komponente ermöglichen und zum anderen aus Synchronisationskanälen, die eine Kopplung des Kontrollflusses zweier Komponenten durch einen Rendevouz-Mechanismus ermöglichen. Im Gegensatz zu synchronen ermöglichen asynchronen Verbindungskanäle einen Datenaustausch während die Komponente sich im Ausführungszustand befindet, d.h. während deren Prozessfragment interpretiert wird.

Die Komposition von Prozesskomponenten erfolgt innerhalb von *Ausführungssichten*, die ähnliche Verbindungsschnittstellen haben wie Prozesskomponenten selbst. Eine Ausführungssicht kann wiederum in anderen Ausführungssichten als Komponenten verwendet werden, so dass dadurch ein komplexes Prozessmodell modular aus Komponenten aggregiert werden kann. Die Instanziierung einer Ausführungssicht erfordert ein schrittweises Einrichten von Verbindungs- und Synchronisationskanälen, Hinzufügen von Prozesskomponenten und Kopplung der Verbindungsschnittstellen der einzelnen Komponenten. Die in einer Ausführungssicht enthaltenen Komponenten und deren Kopplung über Verbindungs- und Synchronisationskanäle können dynamisch durch Funktionen wie z.B. *addComponent* oder *modifyConnection* zur Laufzeit modifiziert werden.

6.2.3.3 Rollenkooperation

In [ShWa95; Warb90] wird ein Prozessmodell durch kooperierende Objekte und Rollen dargestellt. Generell werden Entitäten der realen Welt durch Objekte repräsentiert, die eine oder mehrere Rolle einnehmen oder von einer Rolle benutzt werden, um das Rollenziel zu erreichen; z.B. nimmt ein Objekt Person die Rolle Projektmanager ein, die zum Erreichen des Rollenziels Projektplanung die Objekte Person, Dokument und Werkzeug benötigt.

Rollen und Objekte sind Objekte im objektorientiertem Sinn und werden für jedes Prozessmodell durch Analysemethoden wie CRC-Karten identifiziert; die Methoden einer Rolle sind die Aktivitäten, die in einer definierten Reihenfolge ausgeführt werden müssen, damit das Rollenziel erreicht wird. Die Ausführung der Aktivitäten einer Rolle führt zu Modifikationen der Objekte und erfordert in der Regel eine Kooperation und Interaktion mit weiteren Rollen.

Die Kooperation der einzelnen Rollen eines Prozessmodells wird durch ein übergreifendes Petrinetz modelliert, deren Transitionen die Methoden einer spezifischen Rolle repräsentieren. Die Verbindung zweier Transitionen durch Stellen und Kanten modelliert den Nachrichtenaustausch zwischen Rollen. Die einzelnen Rollen eines Prozessmodells, repräsentiert durch Objektklassen, bilden insgesamt

ein Framework, in dem der Kontrollfluss der Aktivitäten, d.h. die Reihenfolge der Methodenaufrufe, durch das rollenübergreifende Petrinetz spezifiziert wird.

6.2.4 Diskussion und Schlussfolgerungen

Der Literaturüberblick in diesem Abschnitt hat drei wesentliche *Entwurfsalternativen* für die Interoperabilität von Prozessmodellierungssprachen gezeigt. Die Überbrückung der Heterogenität verschiedensprachlich definierter Prozessfragmente kann entweder durch einen *Transformationsansatz*, einen *zentralisierten Prozesszustand* oder einen *Schnittstellenansatz* gelöst werden (siehe Abb. 25).

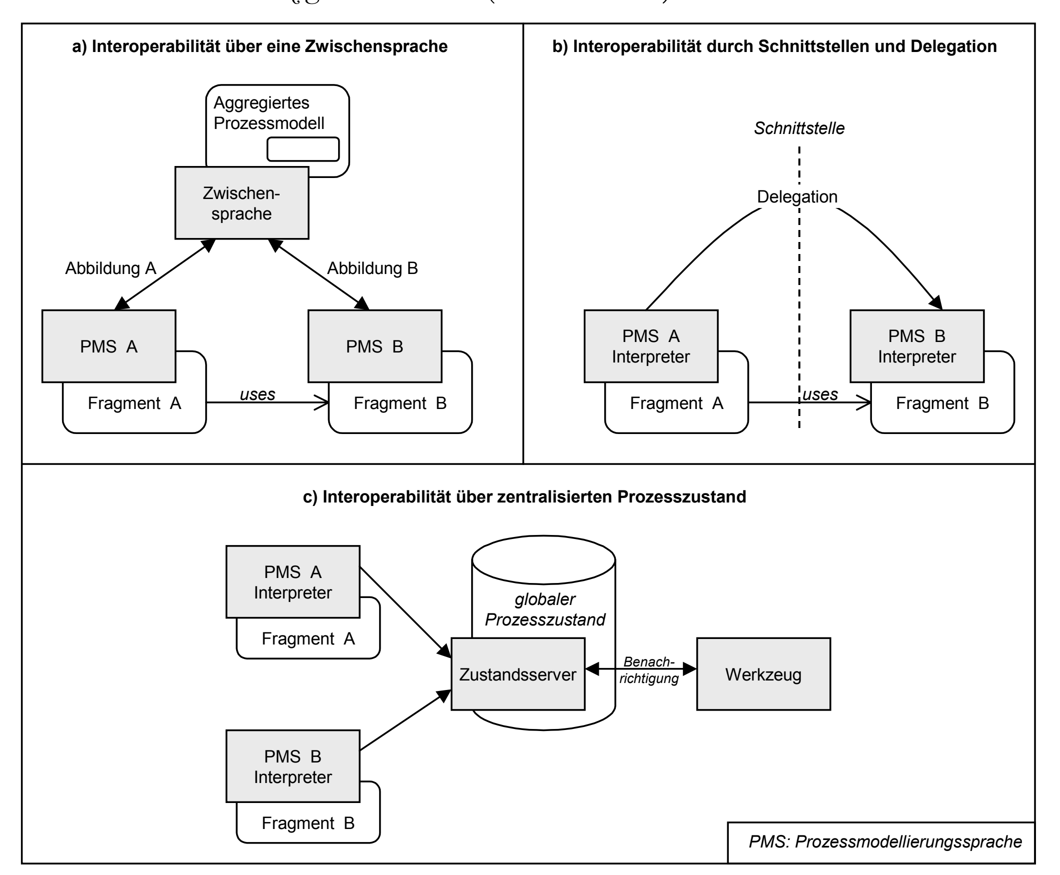


***Abb. 25**: Interoperabilitätsansätze im Vergleich*

Interoperabilität durch Transformation

Bei einem Transformationsansatz wird die Integration der Prozessmodellierungssprachen durch eine gemeinsame Zwischensprache wie z.B. WPDL, PIF oder PSL realisiert, die allgemeine Konzepte von Modellierungssprachen subsummiert (siehe Abschnitt 6.2.1.2 und Abb. 25a). Die Integration erfolgt durch die Definition von Transformationsbeziehungen zwischen den Konzepten der zu integrierenden Prozessmodellierungssprachen und der Zwischensprache. Bei Verwendung einer Fragmentes wird die Zwischensprache dafür genutzt, das Fragment in die Zielsprache des verwendenden Fragmentes zu übersetzten. Verwendet ein Fragment A ein Fragment B, dann wird das Fragment B vollständig durch Transformation über die Zwischensprache in die Sprache A übersetzt. Der Integrationspunkt liegt beim Transformationsansatz in der *Modellierungsdomäne*, so dass die Interoperabilität von Prozesssprachen-Interpretern in der Leitdomäne eine untergeordnete Rolle spielt. Der Hauptvorteil dieser Vorgehensweise ist darin zu sehen, dass die Zwischensprache die einzelnen Formalismen vereinheitlicht und somit verschiedensprachliche Teilprozesse in einem übergreifenden Gesamtprozess einheitlich dargestellt,

analysiert und administriert werden können (vgl. auch WfMC-Interoperabilitätsstufe 6, Abschnitt 6.2.1.1). Die Annahme, dass die Zwischensprache in der Lage ist, alle Prozessmodellierungskonzepte zu vereinigen, ist jedoch idealisiert [LeYo94; WfMC98b]. Daher bieten Austauschformate in der Regel Mechanismen an, mit denen die Konzepte der Zwischensprache erweitert und an die jeweiligen Bedürfnisse angepasst werden können. Insgesamt müssen die zu integrierenden Sprachen jedoch eine weitestgehend mit der Zwischensprache vergleichbare Ausdrucksstärke haben, um eine Transformation zu ermöglichen. Die Option, Interoperabilität von Sprachen mit Hilfe eines gemeinsamen Repräsentationsformates bzw. eines Übersetzungsmechanismus zu erzielen, ist aufgrund der schlechten Skalierbarkeit hinsichtlich heterogener Formalismen daher als kritisch zu bewerten [GaLK98].

Interoperabilität über zentralisierten Prozesszustand

Ein weiterer Ansatz besteht darin, die Interoperabilität über einen zentralisierten Prozesszustand in der Leitdomäne zu erzielen (Abschnitt 6.2.2 und Abb. 25c). Im Gegensatz zur statischen Zwischensprache spiegelt der zentralisierte Prozesszustand die dynamische Ausführung eines instanziierten Softwareprozesses zur Laufzeit wider. Damit ist zwar die Interoperabilität von Prozessmaschinen in der Leitdomäne gewährleistet, jedoch ist unklar, wie die Formalismen in der Modellierungsdomäne interoperabel verwendet werden können. Lediglich die Modifikation des Prozesszustandes, d.h. die Auswirkung der Prozessfragmentinterpretation, wird integriert und nicht die Modellierungssprachen selbst. Da wir jedoch die Aggregation heterogen spezifizierter Plankontexte gerade auch *modellierungsseitig* beschreiben wollen, scheidet diese Option aus.

Interoperabilität über Schnittstellen

Die beim Transformationsansatz gegebene Abhängigkeit von einer einheitlichen Zwischensprache wird durch einen komponentenbasierten Modellierungsansatz überwunden, der die einzelnen Formalismen durch *definierte Schnittstellen* verbirgt (siehe Abschnitt 6.2.3 und Abb. 25b). Um eine Prozesskomponente innerhalb einer anderen verwenden zu können, muss lediglich die Komponentenschnittstelle mit den Konzepten der jeweiligen Verwendungssprache repräsentiert werden, so dass die abgebildete Stellvertreterschnittstelle in der Verwendungssprache angesprochen werden kann. Die Abbildung der Komponentenschnittstelle ist mit der Transformation der Prozessmodelle mit Hilfe einer Zwischensprache vergleichbar, wobei die Zwischensprache dabei der Schnittstellenbeschreibungssprache entspricht. Anstelle jedoch eine vollständige Transformation des zu verwendenden Prozessfragmentes vorzunehmen, ist hier lediglich die Abbildung der Komponentenschnittstelle erforderlich. Semantische Verluste, die z.B. bei der Transformation der Kontrollmodelle der jeweiligen Sprachen auftreten können, werden dadurch vermieden. Dieser Ansatz ist somit hinsichtlich der Integration von Modellierungssprachen wesentlich skalierbarer als ein Transformationsansatz, da die Definition von sprachspezifischen Abbildungsbeziehungen sich auf die für die Darstellung der Schnittstelle notwendigen Informationen beschränkt.

Anders als beim Transformationsansatz ist beim Schnittstellenansatz auch eine Laufzeitunterstützung bei der Kooperation der jeweiligen Prozesssprachen-Interpreter erforderlich. Wenn ein Komponente A ein Komponente B verwendet, dann wird an der Stelle der Verwendung die Ausführung an den Interpreter der Sprache B delegiert. Für die Integration von Prozesssprachen-Interpretern in der Leitdomäne können die von der WfMC oder OMG entwickelten Standards verwendet werden (siehe Abschnitt 6.2.1).

Der wesentliche Nachteil eines komponentenbasierten Ansatzes besteht darin, dass der Verzicht auf einen einheitlichen Basisformalismus die formale Analyse kompositer Prozessfragmente erschwert. Es ist jedoch zu erwarten, dass sich das Verhalten eines kompositen Prozessfragments hinsichtlich bestimmter *kompositionaler* Eigenschaften (z.B. Erreichbarkeit des Endzustands) aus den Eigenschaften der Komponenten ableiten lässt. Auf eine formale Untersuchung solcher Fragestellungen müssen wir jedoch im Rahmen dieser Arbeit aus Aufwandsgründen verzichten.

Da bei uns weniger die Analyse als vielmehr die Ausführung von Prozessfragmenten im Vordergrund steht, ziehen wir in dieser Arbeit einen auf Schnittstellen basierenden Ansatz einem Transformationsansatz vor. Im folgenden Abschnitt erarbeiten wir dazu eine komponentenbasierte Darstellung des NATURE-Prozessmodells, mit der Kontexte als Komponenten sprachübergreifend verwendbar sind.

6.3 Komponentenorientierte Darstellung des NATURE-Prozessmodells

Ein komponentenbasierter Prozessmodellierungsansatz, der die Verwendung von mehreren Prozessmodellmodellierungssprachen und die Komposition eines Prozessmodells aus verschiedensprachlichen Komponenten ermöglicht, muss grundsätzlich zwischen dem *Definitionspunkt* einer Komponente und deren *Verwendungspunkten* unterscheiden. Damit eine Prozesskomponente des Definitionspunkt sprachübergreifend in der Modellierungsdomäne eingesetzt werden kann, muss bei deren Verwendung stets eine Repräsentation in der Prozessmodellierungssprache des Verwendungspunkts gefunden werden, damit die Prozesskomponente dort syntaktisch referenziert und an den Daten- und Kontrollfluss angeschlossen werden kann. Eine sprachneutrale Schnittstellenbeschreibung sowie Transformationsregeln für die Abbildung von Komponentenschnittstellen in die Verwendungssprachen sind hierfür Voraussetzung (siehe Abb. 26).

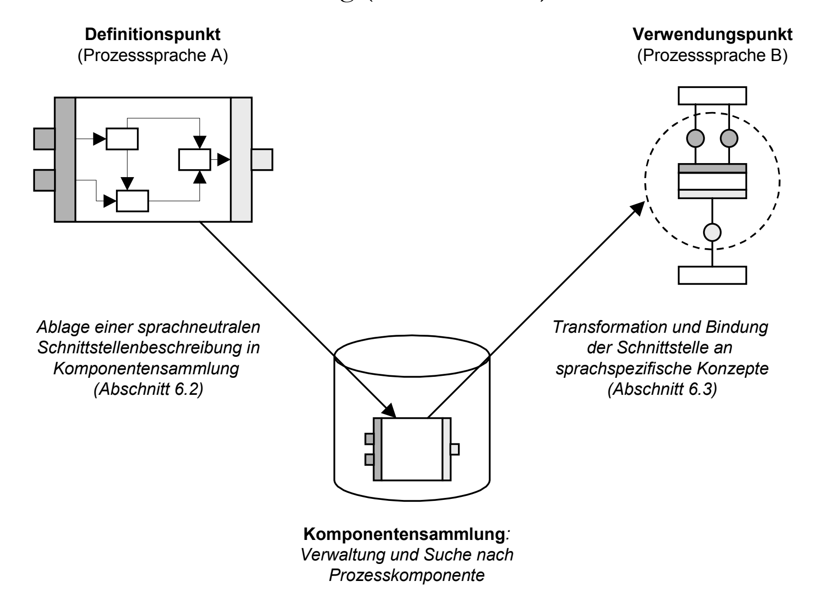


***Abb. 26**: Komponentendefinition und -verwendung*

Analog zu komponentenorientierten Ansätzen aus dem programmiersprachlichen Bereich (z.B. CORBA, COM, DCE oder SLI) weist unser Ansatz zur Interoperabilität verschiedensprachlicher Prozesskomponenten somit drei wesentliche Charakteristika auf:

- Es existiert eine strikte Trennung zwischen der Schnittstelle einer Prozesskomponente und ihrer Implementierung.
- Während die Implementierung einer Prozesskomponente in einer potenziell beliebigen, dafür geeigneten Sprache erfolgt, liegen für alle Prozesskomponenten Schnittstellenbeschreibungen in einem einheitlichen, sprachneutralen Format vor.
- Prozessfragmentkomponenten werden in einer Komponentensammlung abgelegt. Dort können sie über ihre neutrale Schnittstellenbeschreibung aufgefunden werden und in anderen Prozesskomponenten wiederverwendet werden.

In diesem Abschnitt befassen wir uns mit der Separation von Schnittstellen- und Implementierungsaspekten und entwickeln ein Metamodell zur neutralen Schnittstellenbeschreibung von Prozesskomponenten, das die Grundlage eines Komponenten-Repositories bildet. In Abschnitt 6.4 betrachten wir die Transformation und Bindung der Schnittstellen von Prozesskomponenten in spezifischen Prozessmodellierungssprachen. Formal gehen wir dabei so vor, dass wir zunächst Metamodelle zur Schnittstellenbeschreibung und für spezifische Prozessmodellierungssprachen in Form von Klassenmodellen spezifizieren. Die prinzipiell möglichen Bindungen zwischen sprachneutralen Schnittstellenkonzepten und sprachspezifischen Implementierungskonstrukten werden einerseits über ein gemeinsames Metametamodell und andererseits über ein explizites Bindungsmetamodell festgelegt. Mithilfe von O-Telos-Integritätsbedingungen werden weitere Eigenschaften einer korrekten Bindung zwischen Schnittstellendefinitionen und sprachspezifischen Konstrukten auf der Instanzebene zugesichert.

Formaler Ansatz

6.3.1 Prozesskomponenten

Wie bereits eingangs dieses Kapitels dargestellt, besteht ein Kontext als Abstraktion eines Prozessfragments aus drei unterschiedlichen Komponenten: einer Intentionskomponente, einer Situationskomponente und einer Verhaltenskomponente. In dieser Arbeit bezeichnen wir Situationskomponenten, Intentionskomponenten, Verhaltenskomponenten und daraus aggregierte Kontextkomponenten verallgemeinernd als *Prozesskomponenten*. Dabei sind die Intentionskomponente sowie die Verhaltenskomponente von Ausführungs- und Entscheidungskontexten durch das Umgebungsmetamodell für eine Operationalisierung bereits vollständig spezifiziert, während die Spezifikation von Situationskomponenten sowie von Abläufen innerhalb von Plankontexten in einer spezifischen Sprache erfolgen kann und durch neutrale Schnittstellen zu kapseln ist.

Abb. 27 illustriert, welche Anteile einer Prozesskomponente der Schnittstellenbeschreibung bzw. der Implementierungsdefinition zuzurechnen sind. Eine Kontextkomponente setzt sich den drei Teilkomponenten Verhalten, Situation

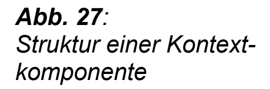
und Intention[18] zusammen. Der Situationsteil und der Verhaltensteil stellen selbst wiederverwendbare Komponenten mit einer definierten Ein- und Ausgangsschnittstelle dar.

Situationskomponente

Die *Eingangsschnittstelle* einer Situationskomponente referenziert den Ausschnitt aus dem Produktmodell, auf dem die Situation auszuwerten ist. Dabei wird die Eingangsschnittstelle aus einer Menge von getypten *Produktteilen* gebildet, denen jeweils spezifische *Rollenbezeichner* zugewiesen werden können. Die Auswertung der Situation erfolgt mithilfe eines *Situationsausdruck*, der die Implementierung einer Situationsspezifikation darstellt. Der Situationsausdruck formuliert in einer dafür geeigneten Sprache Bedingungen, die erfüllt sein müssen, damit die an der Eingangsschnittstelle vorliegenden Produkte eine gültige Situation bilden. Die Ausgangsschnittstelle einer Situation besteht wiederum aus Produktteilen in spezifischen Rollen. Diese Produktteile bilden zusammen die Produktkonstellationen, die den Situationsausdruck erfüllen. Produktteile der Eingangsschnittstelle nennen wir *In-Produktteile*, während wir Datenbehälter der Ausgangsschnittstelle als *Out-Produktteile* bezeichnen. Der Bezug zwischen den In- und Out-Produktteilen einer Situationskomponente hängt von der Art des Situationsausdrucks an, der entweder einen *filternden* oder *transformierenden* Charakter haben kann. Im ersten Fall werden die an der Eingangsschnittstelle vorliegenden Produktteile einfach an die Ausgangsschnittstelle weitergereicht, sofern sie den Bedingungen des Situationsausdrucks genügen. Die Ein- und Ausgangsschnittstelle haben dann eine identische Struktur, d.h. sie bestehen aus einer gleichen Anzahl von In- und Out-Produktteilen, die in Rolle und Typ jeweils paarweise übereinstimmen. Bei einem transformierenden Situationsausdruck referenzieren die In-Produktteile einen bestimmten Ausschnitt des Produktmodells, innerhalb dessen dann die Produktkonstellationen ermittelt werden, die den Situationsausdruck erfüllen.

Abb. 27 zeigt ein Beispiel für eine solche Situationskomponente: die Situation `UnconnectedEntities` soll für alle Entitäten `e` innerhalb eines ER-Diagramms `erd` gültig sein, die nicht mit einer anderen Entität über eine Beziehung verbunden sind. In diesem Fall referenziert das Produkt `erd` den aktuell relevanten Produktausschnitt (ein ER-Diagramm inklusive aller darin enthaltenen Elemente wie Entitäten, Beziehungen, Attribute etc.). Der Situationsausdruck wird durch eine entsprechende SQL-Anfrage auf der Produktdatenbank implementiert. An der Ergebnisschnittstelle liegen die Entitäten des ER-Diagramms `erd` vor, die den Situationsausdruck erfüllen, d.h. die Ergebnisse der Anfrage. Zu beachten ist, dass wir nicht vorschreiben, mit welchem Formalismus der Situationsausdruck spezifiziert wird. In diesem Fall ist dies SQL; in Kapitel 7 stellen wir eine einfache Situationssprache vor, die über dem aktuellen Zustand der in einem Werkzeug geladenen Produkten ausgewertet wird.

[18] Wir ignorieren hier und in der weiteren Diskussion den Intentionsteil eines Kontext, da wir eine Intention lediglich durch ihren Bezeichner repräsentieren und keine weitere Substruktur, d.h. eine Implementierung, des Intentionsbegriffs voraussetzen.



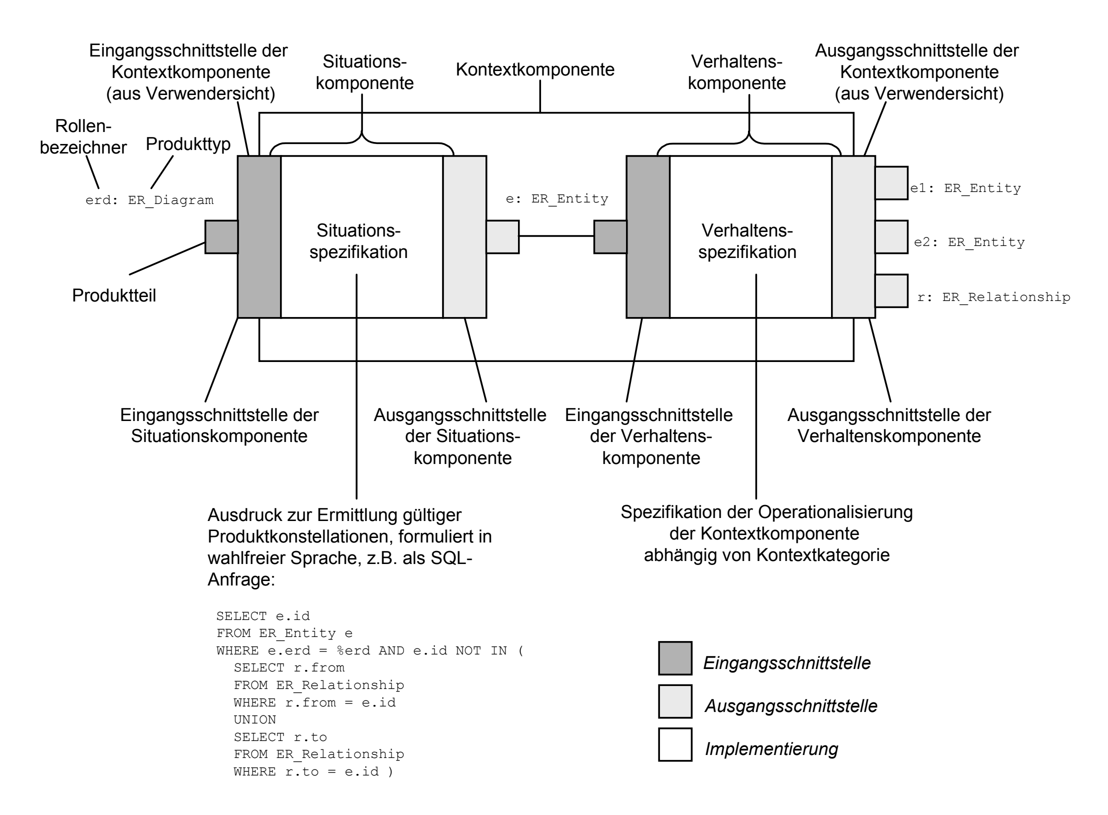


***Abb. 27**: Struktur einer Kontext-komponente*

Verhaltenskomponente

Ebenso wie Situationskomponenten verfügen Verhaltenskomponenten über eine Ein- und Ausgangsschnittstelle, die aus einer Menge von getypten In- bzw. Out-Produktteilen besteht. Verhaltenskomponenten haben einen transformierenden Charakter, so dass grundsätzlich keinerlei struktureller Bezug zwischen den Ein- und Ausgangsschnittstellen gefordert wird. Wie bereits eingangs dieses Kapitels erläutert, hängt die Implementierung der Verhaltenskomponente von der Kontextkategorie ab und ist bei Ausführungs- und Entscheidungskontexten schon vollständig auf der Ebene des Umgebungsmetamodells spezifiziert. Bei Plankontexten stellt die Spezifikation einer Ablaufreihenfolge in einer wahlfreien Sprache die Implementierung dar.

Kontextkomponente

Eine Kontextkomponente aggregiert eine Situationskomponente und eine Verhaltenskomponente. Die aus Verwenderperspektive sichtbare äußere Schnittstelle einer Kontextkomponente wird durch die Eingangsschnittstelle der Situationskomponente und die Ausgangsschnittstelle der Verhaltenskomponente gebildet (siehe Abb. 27). Intern werden Situations- und Verhaltenkomponente miteinander verknüpft, indem die Out-Produktteile der Situationskomponente auf die In-Produktteile der Verhaltenkomponente abgebildet werden. Die Ausgangsschnittstelle der Situationskomponente muss dabei die Eingangsschnittstelle der Verhaltenskomponente subsummieren, d.h. jedem In-Produktteil der Verhaltenskomponente muss jeweils ein Out-Produktteil der Situationskomponente zugeordnet werden,

wobei die miteinander verknüpften Produktteile jeweils den gleichen Typ haben[19]. Es wird jedoch nicht gefordert, dass alle Out-Produktteile der Situationskomponente vollständig auf In-Datenbehälter der Verhaltenskomponente abgebildet werden müssen. Dies erlaubt eine flexiblere Verknüpfung von Situations- und Verhaltenskomponenten auch in solchen Fällen, in denen einige der Out-Produktteile der Situationskomponente für die Verhaltenskomponente irrelevant sind.

6.3.2 Schnittstellenmetamodell

Nachdem wir im voran gegangenen Unterabschnitt auf informeller Ebene die Schnittstellen- und Implementierungsanteile von Kontextkomponenten vorgestellt haben, formulieren wir nun ein formales Metamodell zur Schnittstellenbeschreibung, das alle aus Verwendungssicht erforderlichen Informationen über eine Kontextkomponente bereitstellt.

Da das Ziel in der Einbettung existierender Prozessmodellierungssprachen in das NATURE-Prozessmodell besteht, orientieren sich die Konzepte zur Schnittstellenbeschreibung naturgemäß an dem durch das NATURE-Prozessmodell vorgegebenen Rahmen. Als Schnittstellenbeschreibungssprache ist das NATURE-Prozessmodell bzw. das Umgebungsmetamodell jedoch in seiner bisherigen Form aus folgenden Gründen ungeeignet:

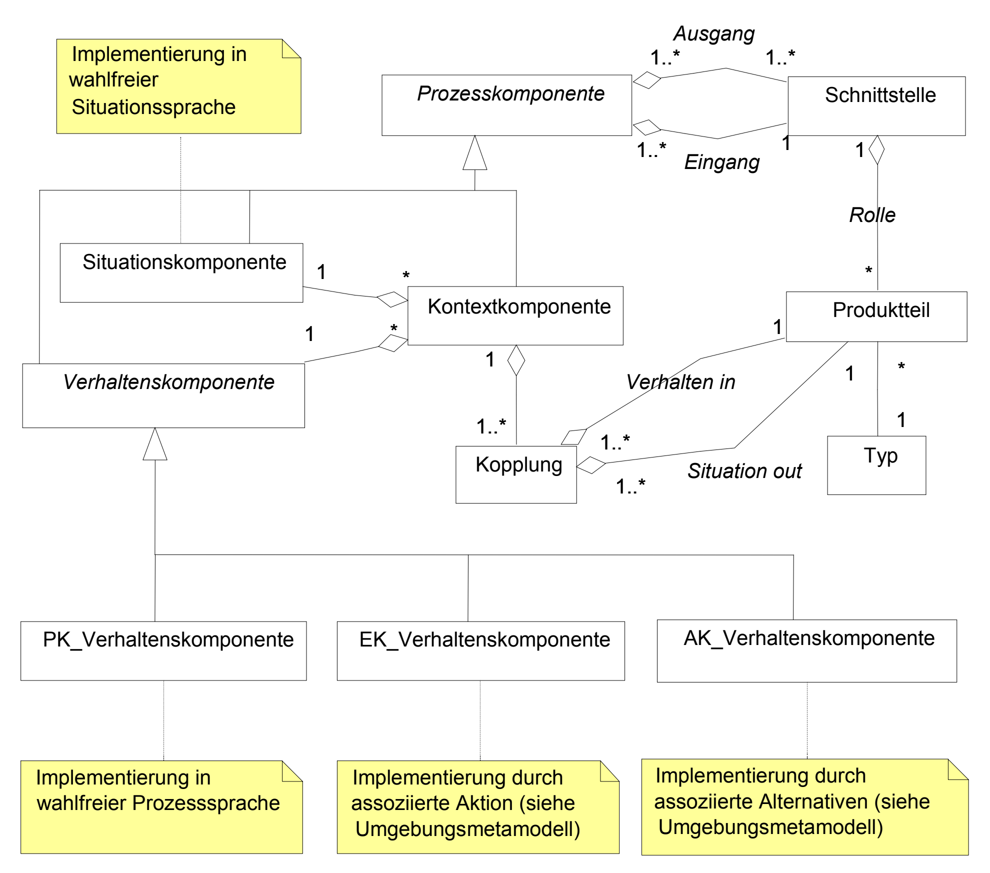
- Schnittstellen- und Implementierungsaspekte werden vermischt. Beispielsweise ist es aus Sicht eines Verwenders unerheblich, durch welche Aktion ein Ausführungskontext operationalisiert wird. Solche Informationen sollte daher nicht in einem Metamodell zur Schnittstellenbeschreibung auftauchen bzw. davon separiert werden.
- Kontexte verfügen in Form der assoziierten Situation zwar über eine *Eingangsschnittstelle*, nicht jedoch über eine explizit definierte *Ausgangsschnittstelle*. Die Auswirkungen eines Kontexts sind im Falle eines Ausführungskontexts lediglich implizit über die Ausgangsprodukte der assoziierten Aktion definiert. Noch weniger offensichtlich sind die Ausgangsschnittstellen von Entscheidungs- und Plankontexten, die sich allenfalls aus den Effekten der Alternativ- bzw. Subkontexte rekursiv ableiten lassen. Die Änderung eines globalen Datenbestand und die dadurch induzierte implizite Aktivierung neuer Kontexte entspricht der regel- oder auslöserbasierten Grundphilosophie des NATURE-Prozessmodells, ist jedoch für die Definition von Plankontexten ungeeignet, da Plankontexte ja gerade eine explizite, präskriptive Ablaufreihenfolge über eine Menge von Subkontexten legen sollen. Insbesondere wird dabei der Kontroll- und Datenfluss zwischen den Kontexten festgelegt, was neben einer Eingangsschnittstelle auch die Definition einer expliziten Ausgangsschnittstelle erfordert.

Wir haben daher das NATURE-Prozessmodell zu einem Metamodell zur Schnittstellenbeschreibung von Kontextkomponenten umformuliert und erweitert (siehe Abb. 28). Das zentrale Element des *Schnittstellenmetamodells*, das vollständig kompatibel zum NATURE-Prozessmodell ist, ist die `Prozesskomponente`. Eine Prozesskomponente verfügt über eine Eingangs- und Ausgangsschnittstelle, was durch

[19] Diese Bedingung verallgemeinert die Konsistenzbedingung AK2, die wir bereits in Abschnitt 5.5.2.1 für den Spezialfall von Ausführungskontexten formuliert hatten.

entsprechende Assoziationen zur Klasse Schnittstelle dargestellt ist. Eine Schnittstelle selbst aggregiert einen oder mehrere Produktteile in jeweils spezifischen Rollen. Die Datenbehälter sind durch ihren Typ näher beschrieben. Die Klasse Prozesskomponente generalisiert die Klassen Kontextkomponente, Situationskomponente und Verhaltenskomponente und vererbt diesen Klassen somit die Fähigkeit, eine Eingangs- und Ausgangsschnittstelle zu besitzen. Eine Verhaltenskomponente unterteilt sich entsprechend dem NATURE-Prozessmodell weiter in die Klassen Plankontext_Verhaltenskomponente, Ausführungskontext_Verhaltenskomponente und Entscheidungskontext_Verhaltenskomponente. Situations- und Verhaltenskomponenten verfügen über spezifische Implementierungsinformationen, die jedoch auf der Ebene der Schnittstellenbeschreibung nicht von Belang sind. Eine Kontextkomponente aggregiert genau eine Situationskomponente und eine Verhaltenskomponente. Je nach Kategorie der an einer Kontextkomponente beteiligten Verhaltenkomponente unterscheiden wir die Klassen Ausführungskontextkomponente (kurz: AK-Komponente), Entscheidungskontextkomponente (kurz: EK-Komponente) und Plankontextkomponente (kurz: PK-Komponente) als Spezialisierung der Klasse Kontextkomponente (in Abb. 28 nicht explizit dargestellt).

***Abb. 28**: Schnittstellenmetamodell*



Bei der Verknüpfung einer Situations- mit einer Verhaltenskomponente wird eine Zuordnung zwischen den Out-Produktteilen der Situationskomponente und den In-Produktteilen der Verhaltenskomponente mithilfe der Klasse Kopplung definiert. Die bereits oben informell skizzierten Konsistenzbedingungen, die für eine korrekte Kopplung gelten müssen, können wir nun als O-Telos-Integritätsbedingungen formalisieren.

```
MetaClass Kontextkomponente isA Prozesskomponente with
  constraint
    korrekteKopplung : $
      forall kk/Kontextkomponente sk/Situationskomponente
             sa/Schnittstelle pta/Produktteil t/Typ
        ( (kk hat_Situationskomponente sk) and(sk Ausgang sa) and
          (sa hat_Produktteil pta) and (pta hat_Typ t) )
        ==>
        ( exists k/Kopplung vk/Verhaltenskomponente
                 se/Schnittstelle pte/Produktteil
             ( (kk hat_Verhaltenskomponente vk) and (vk Eingang se) and
               (se hat_Produktteil pte) and (kk hat_Kopplung k) and
               (k Situation_out pta) and (k Verhalten_in pte) and
               (se hat_Typ t) ) ) $
end
```

Aus Verwendungssicht ist nur die *äußere Schnittstelle* einer Kontextkomponente relevant. Diese ergibt sich aus der Eingangsschnittstelle der Situationskomponente und der Ausgangsschnittstelle der Verhaltenskomponente und kann durch zwei O-Telos-Regeln (berechneEingang und berechneAusgang) hergeleitet werden.

```
MetaClass Kontextkomponente isA Prozesskomponente with
  attribute, necessary
    hat_Situationskomponente : Situationskomponente;
    hat_Verhaltenskomponente : Verhaltenskomponente
  rule
    berechneEingang : $
      forall sk/Situationskomponente s/Schnittstelle
        (this hat_Situationskomponente sk) and (sk Eingang s)
        ==> (this Eingang s) $
    berechneAusgang : $
      forall vk/Verhaltenskomponente s/Schnittstelle
        (this hat_Verhaltenskomponente vk) and (vk Ausgang s) )
        ==> (this Ausgang s) $
end
```

Modellierung des Typsystems

Bei der Modellierung des Typsystems nehmen wir an, dass sich die zu integrierenden Modellierungssprachen und das Schnittstellenmetamodell ein gemeinsames Typsystem zur Definition des Produktmodells teilen. Diese Vorgehensweise ist naheliegend, da wir gemäß der Anforderung nach Datenintegration davon ausgehen, dass das zugrunde liegende Produktmodell über die gesamte Entwurfsumgebung integriert ist und zentral von Produktmodellierungswerkzeugen bearbeitet werden kann. Ein Typ wird somit einmalig global spezifiziert und kann dann in jeder für die Implementierung von PK-Komponenten benutzten Prozessmodellsprache verwendet werden. Dies stellt zwar eine Vereinfachung dar, da genaugenommen jede Sprache ihr eigenes Typsystem mit Basis- und benutzerdefinierten Typen mitbringt. Die Abbildung von IDL-Spezifikationen auf sprachspezifische Typsysteme in Ansätzen wie CORBA oder COM zeigt jedoch, dass dies grundsätzlich ohne schwerwiegende semantische Verluste möglich ist, so dass wir die Interoperabilität von Typspezifikationen zur Produktmodellierung hier nicht weiter untersuchen.

Ausgangsschnittstelle von Entscheidungskontext-Komponenten

Etwas komplizierter als bei Ausführungs- und Plankontexten ist die Beschreibung der Ausgangsschnittstelle von Entscheidungskontext-Komponenten (EK-Komponenten), die den Benutzer bei der Auswahl unter einer Menge von Vorgehensalternativen beraten. Hier muss zunächst geklärt werden, was unter der *Ausführung* einer EK-Komponente überhaupt zu verstehen ist. Prinzipiell sind zwei Interpretationsmöglichkeiten denkbar.

Zum einen könnte man unter der Ausführung einer EK-Komponente lediglich den Dienst der Auswahl einer Alternative durch den Benutzer, nicht aber deren Ausführung verstehen. Diese Sichtweise, die in einer früheren Version des PRIME-Rahmenwerks eingenommen wurde, erwies sich aus einer Reihe von Gründen als wenig vorteilhaft:

- An jedem Verwendungspunkt einer EK-Komponente muss die anschließende Ausführung der gewählten Alternative explizit spezifiziert werden. Da die Auswahl zudem nichtdeterministisch vom Benutzer abhängt, müssen jeweils alle potenziellen Alternativen berücksichtigt werden. Dies ist aufwändig und auch fehlerträchtig: wenn eine Alternative zu einer EK-Komponente hinzugefügt oder aus dieser entfernt wird, müssen alle PK-Komponenten, die die EK-Komponente verwenden, entsprechend angepasst werden.
- Da die Alternativen einer EK-Komponente am Verwendungspunkt direkt sichtbar sind, muss die Alternativenauswahl in der spezifischen Prozesssprache der verwendenden PK-Komponente dargestellt werden. Hier ergibt sich jedoch die Gefahr semantischer Unterschiede, wenn die Alternativen z.B. in imperativen Sprachen durch bedingte Verzweigungen und in Petrinetzsprachen durch parallele Transitionen dargestellt würden. Damit wäre jedoch die Semantik einer EK-Komponente abhängig vom Verwendungspunkt, da bedingte Verzweigung und Parallelität eine unterschiedliche Bedeutung haben.
- EK-Komponenten einerseits und AK- bzw. PK-Komponenten andererseits weisen eine unterschiedliche Schnittstellenstruktur auf. Anders als bei AK- und PK-Komponenten würde die Ausgangsschnittstelle einer EK-Komponente nicht aus einer Menge getypter Produktteile für die erzeugten oder modifizierten Produkte bestehen. Stattdessen wäre das Resultat der Ausführung einer EK-Komponente die zur Laufzeit ausgewählte Instanz einer Kontextkomponente und die Ausgangsschnittstelle hätte somit den Typ `Kontextkomponente`.

Sinnvoller erscheint daher eine zweite Interpretationsmöglichkeit, bei der der Ausführungsdienst einer EK-Komponente die Benutzerauswahl einer Alternative *und* deren Ausführung vollständig kapselt. Aus Verwendersicht ist demnach nur die Ein- und Ausgangsschnittstelle einer EK-Komponente sichtbar. Diese Sichtweise hat jedoch zwei wichtige Folgerungen:

- Am Verwendungspunkt einer EK-Komponente ist nicht erkennbar, welche Alternative tatsächlich ausgewählt und ausgeführt wird. Eine EK-Komponente muss also zur Laufzeit durch jede ihrer Alternativen *substituierbar* sein, und es ist für die weitere Prozessdurchführung unerheblich, welche Alternative ausgewählt und ausgeführt wurde.

- Um Substituierbarkeit zu garantieren, müssen eine EK-Komponente und ihre Alternativen jeweils zueinander schnittstellenkompatibel sein. Schnittstellenkompatibilität ist hier analog zur Verfeinerung von Methodensignaturen in stark typisierten, objektorientierten Sprachen zu sehen[20], d.h. die Ein- und Ausgangsschnittstellen der EK-Komponente und ihrer Alternativen müssen den gleichen strukturellen Aufbau haben und die In-Produktteile der Alternativen müssen dabei den gleichen oder einen allgemeineren Typ, die Out-Produktteile den gleichen oder einen spezielleren Typ als die entsprechenden Produktteile der EK-Komponente tragen. Außerdem muss die Situationsbedingung der Alternativen schwächer sein als die der EK-Komponente.

Durch diese Forderungen entsteht eine Spezialisierungsbeziehung zwischen Kontextkomponenten, die festlegt, ob eine Kontextkomponente als Alternative einer EK-Komponente auftreten darf. Die so entstehenden Hierarchien sind jedoch unserer Erfahrung nach relativ flach, da nur eng verwandte Kontextkomponenten in einer Spezialisierungsbeziehung angeordnet werden können. Diese Erfahrungen wurden auch durch ähnliche Arbeiten im Bereich der Ablaufmodellierung für verfahrenstechnische Entwurfsprozesse bestätigt [Lohm98; Krob97]. Insgesamt wird dadurch die Flexibilität des Methodeningenieurs bei der Modellierung von EK-Komponenten stark eingeschränkt.

Vor dem Hintergrund dieser Überlegungen haben wir uns bei der Schnittstellenmodellierung von EK-Komponenten zu einer Kompromisslösung entschlossen, die die Nachteile der oben skizzierten Ansätze weitestgehend vermeidet. Hierbei schließt die Ausführung einer EK-Komponente wie beim letztgenannten Lösungsansatz nicht nur die Benutzerauswahl, sondern auch die Ausführung der gewählten Alternative ein. Wir verlangen jedoch nicht mehr, dass die Ausgangsschnittstellen der Alternativen jeweils eine Spezialisierung einer einzigen Ausgangsschnittstelle der übergeordneten EK-Komponente darstellen müssen. Stattdessen erlauben wir, dass aus Sicht einer verwendenden PK-Komponente eine EK-Komponente nicht nur eine, sondern mehrere Ausgangsschnittstellen besitzen kann, die genau den Ausgangsschnittstellen der Alternativen entsprechen. Je nach Benutzerauswahl wird zur Laufzeit dann die zur ausgewählten Alternative gehörige Ausgangsschnittstelle aktiv. Damit eine EK-Komponente zur Modellierungszeit an den Datenfluss innerhalb einer verwendenden PK-Komponente angeschlossen werden kann, muss garantiert werden, dass sie die Ausgangsschnittstellen aller enthaltenen Alternativen umfasst. Dies lässt sich formal durch folgende O-Telos-Regel sicherstellen:

```
MetaClass EK_Verhaltenskomponente isA Verhaltenskomponente with
  attribute
    alternative : Kontextkomponente
  rule
    berechneAusgang : $
      forall vk / EK_Verhaltenskomponente
             kk / Kontextkomponente s / Schnittstelle
        (vk alternative kk) and (kk Ausgang s)
        ==> (vk Ausgang s) $
end
```

[20] Für eine Diskussion der so genannten Ko-/Kontravarianz-Problematik siehe z.B. [KeMo93].

Neben dem Anschluss der Kontextalternativen an den Datenfluss ist es weiterhin erforderlich, dass der nachfolgende Kontrollfluss in Abhängigkeit von der gewählten Kontextalternative verzweigen kann. Da die Kontextalternativen in der verwendenden PK-Komponente nicht explizit dargestellt werden, muss die Auswahl ebenfalls indirekt an der Ergebnisschnittstelle der CC-Komponente verfügbar sein.

6.3.3 Zusammenfassung

In diesem Abschnitt haben wir auf Basis des NATURE-Prozessmodells ein Metamodell entwickelt, das aus komponentenorientierter Sicht die Schnittstellen von Prozesskomponenten beschreibt. Zusammenfassend besteht eine Kontextkomponente aus einer Situations- und einer Verhaltenkomponente. Die Situationskomponente spezifiziert über einen Situationsausdruck eine Vorbedingung und ist über eine explizite Kopplung ihrer Ausgangsschnittstelle mit der Verhaltenskomponente verknüpft. Sowohl die Situations- als auch die Verhaltenskomponente sind aus Verwendungssicht transparent, da beide der internen Struktur einer Kontextkomponente zuzuordnen sind. Das Dienstangebot der Kontextkomponente ist über die äußere Komponentenschnittstelle verfügbar. Die drei Kontextarten werden indirekt über die Verhaltenskomponente abgebildet, die je nach Kontextart durch eine Aktion, einen Ablauf oder die Angabe der Kontextalternativen implementiert werden.

6.4 Schnittstellenbindung

Das im vorangegangenen Abschnitt vorgestellte Schnittstellenmetamodell ist zunächst unabhängig von einer spezifischen Prozessmodellierungssprache zu betrachten, da es aus Verwendungssicht ähnlich einer IDL-Schnittstellenbeschreibungssprache nur als neutrale Beschreibungssprache für Kontextkomponenten dient. Damit eine so beschriebene Kontextkomponente im Rahmen einer Plankontextspezifikation in einer konkreten Prozesssprache verwendet werden kann, ist es erforderlich, dass das Schnittstellenmetamodell und das Metamodell der gewählten Prozesssprache integriert und über explizite *Bindungen* zueinander in Beziehung gesetzt werden. Eine Bindung drückt aus, wie eine Kontextkomponente in einer gewählten Prozessmodellierungssprache syntaktisch referenziert wird.

Bei unserem Ansatz erfolgt die Bindung zwischen dem Schnittstellenmetamodell und den in spezifischen Kontexten jeweils verwendeten Prozessmodellierungssprache nicht ad-hoc, sondern wird durch ein *Prozesssprachen-Metametamodell* (im Folgenden PSM2-Modell genannt) *systematisch* gesteuert. Das PSM2-Modell, das in Abschnitt 6.4.1 vorgestellt wird, abstrahiert sowohl vom kontextbasierten Schnittstellenmetamodell als auch von den Basiskonzepten einer Prozessmodellierungssprachen und definiert dadurch gleichzeitig die Mindestanforderungen für die Integrationsfähigkeit einer zu integrierenden Sprache. Des weiteren bildet das PSM2-Modell das Fundament für eine Integrationsmethodik, die in Abschnitt 6.4.2 erläutert wird.

6.4.1 M2-Modell

Das M2-Modell besteht aus zwei Submodellen: dem *Prozesssprachen-M2-Modell* (PSM2-Modell) und dem *Bindungs-M2-Modell* (BM2-Modell). Das PSM2-Modell klassifiziert die Sprachkonzepte der zu integrierenden Prozesssprachen. Die Instanziierung des PSM2-Modells sowohl durch das Schnittstellenmetamodell als auch durch das Metamodell der zu integrierenden Sprachen gibt die grundsätzlichen Korrespondenzen zwischen Kontextkomponenten und sprachspezifischen Konzepten bereits vor: nur Sprachkonzepte, die der gleichen Klasse im PSM2-Modell angehören, können aneinander gebunden werden. Die eigentliche Bindung von Sprachkonzepten wird dann innerhalb des BM2-Modells festgelegt.

6.4.1.1 Prozesssprachen-M2-Modell

Das PSM2-Modell abstrahiert diejenigen Sprachkonzepte, die geeignet sind, Kontextkomponenten in einer spezifischen Prozessmodellierungssprache darzustellen (siehe Abb. 29).

Da Kontexte hier als Komponenten verstanden werden, die über Schnittstellen einen Dienst kapseln, ist es naheliegend, die drei Kontextarten einheitlich durch das Konzept einer `Aktivität` darzustellen. Nahezu jede Prozessmodellierungssprache kennt dieses Konzept, das einen atomaren oder zusammengesetzten Prozessschritt repräsentiert. Die Sichtweise, das Dienstangebot einer Kontextkomponente durch das Aktivitätskonzept einer Sprache darzustellen, ist durchaus konsistent mit dem NATURE-Prozessmodell allgemein. Neben Ausführungskontexten und Plankontexten, die hier als automatisierte Werkzeug- bzw. Interpretationsaktivitäten verstanden werden, wird die Zielverfeinerung im Rahmen eines Entscheidungskontexts, d.h. der Entscheidungsvorgang als solcher, ebenfalls als Aktivität aufgefasst.

Wichtig ist, dass das Aktivitätskonzept einer Prozesssprache mit einer `Schnittstelle` einher geht, mit der die *Dekomposition* und *Modularität* von Aktivitäten erzielt werden kann und eine Trennung der Implementierung vom Dienstangebot vorgenommen werden kann.

Der Datenaustausch zwischen einer Aktivität und ihrer Umgebung erfolgt über deren Ein- und Ausgangsschnittstelle. Dazu werden an der Schnittstelle `Datenbehälter` für die Aufnahme von Produkten in einer spezifischen Rolle deklariert. Die Existenz eines Datenbehälterkonzepts in einer Prozessmodellierungssprache garantiert, dass die Aktivität an den umgebenden Datenfluss angeschlossen werden kann und dass die Implementierung von der Aktivitätsschnittstelle getrennt ist.

Aktivitäten sind entweder `atomare Aktivitäten` oder als `zusammengesetzte Aktivitäten` in weitere Subaktivitäten strukturiert. Aus *Verwendungssicht* werden AK-, EK- und PK-Komponenten einheitlich durch das Sprachkonzept einer atomaren Aktivität einer Prozessmodellierungssprache repräsentiert werden. Dazu gehören z.B. Transitionen in Petrinetz-orientierten Sprachen oder einfache Zustände in Zustandsdiagrammen. Die Atomizität bezieht sich dabei lediglich auf die Darstellung der Komponente im Fragment und nicht auf ihre innere Zusammensetzung, denn intern setzen sich Kontextkomponenten selbstverständlich aus weiteren Teilkomponenten zusammen, die jedoch aus Verwendungssicht nicht sichtbar sind.

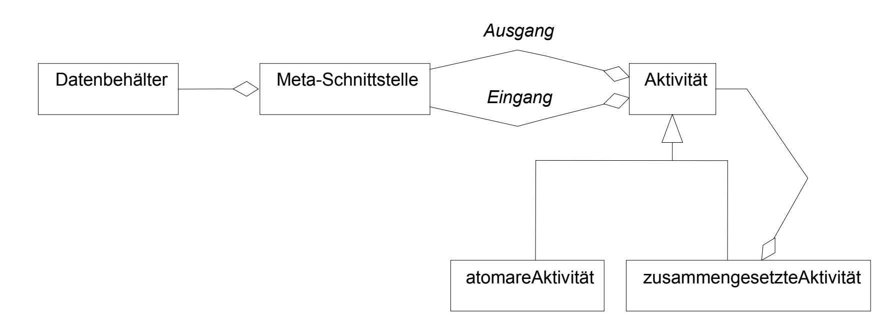


Abb. 29: *Prozesssprachen-M2-Modell*

`Zusammengesetzte Aktivitäten` bilden die Struktur von Modellierungssprachen ab, die eine Dekomposition eines Fragmentes in weitere Teilfragmente ermöglichen und in der sich die *Implementierung* des Fragments auf die atomaren Aktivitäten als Basiselemente stützt. Durch Komposition von atomaren Aktivitäten kann schrittweise der komplexe Dienst des Fragmentes realisiert werden, der über die Aktivitätsschnittstelle in Anspruch genommen wird. Diese Eigenschaft entspricht der Kompositionsstruktur von Plankontexten, die sich ebenfalls aus weiteren Kontexten zusammensetzen, und ist daher Voraussetzung für die Integrierbarkeit einer Sprache.

Neben der Abstraktion der Prozesssprachen-Basiskonzepte abstrahiert das PSM2-Modell auch von den Konzepten des Schnittstellenmetamodells. Dies ist erforderlich, damit bei der Integration einer Sprache die jeweiligen Konzepte der Modelle assoziiert werden können. Die Instanz einer atomaren Aktivität ist im Schnittstellenmetamodell durch eine Kontextkomponente gegeben, da sie analog zu beispielsweise einer SLANG-Transition über ihre Schnittstelle einen Dienst anbietet. PK- und EK-Komponenten sind Instanzen einer `zusammengesetzten Aktivität`, da sich beide aus weiteren Kontexten zusammensetzen. Weiterhin werden auch die Komponentenkonzepte `Schnittstelle` und `Produktteil` als direkte Instanzen der M2-Modellkonzepte dargestellt. Tab. 9 zeigt die Ausprägungen der Konzepte des M2-Modells einiger Sprachen; die korrespondierenden Konzepte des Schnittstellenmetamodells sind kursiv dargestellt.

Tab. 9: *Sprachkonzepte als Instanzen des PSM2-Modells*

	Datenbehälter	**Meta-Schnittstelle**	**Atomare Aktivität**	**Zusammen-gesetzte Aktivität**
Schnittstellen-metamodell	*Produktteil*	*Situationsschnittstelle*	*Kontextkomponente*	*PK_Verhaltens-komponente*
SLANG	Stelle	Vor- bzw. Nachbereich	Transition	Aktivitätsnetz
UML-Zustands-diagramme	Attribut	Aktion	Zustand	Zustandsdiagramm
Funsoft	Kanal	Vor- bzw. Nachbereich	Tätigkeit	Tätigkeitsnetz
HFSP	Attribut	Funktionsschnittstelle	Subfunktion	Funktion

6.4.1.2 Bindungs-M2-Modell

Bei der Verwendung einer Kontextkomponente innerhalb einer PK-Komponente ist es erforderlich, dass diese durch eine *Stellvertreterschnittstelle* in der Syntax der Prozessmodellierungssprache der PK-Komponente dargestellt werden kann, da sie ansonsten mangels einer syntaktischen Repräsentation nicht referenziert werden könnte. Die Regeln für die Erzeugung der Stellvertreterschnittstelle werden hier mit Hilfe von *Bindungen* umgesetzt, die für jede Prozessmodellierungssprache festlegen, wie eine Stellvertreterschnittstelle mit den sprachspezifischen Konzepten dargestellt wird. Die prinzipielle Vorgehensweise entspricht dem CORBA- bzw.

SLI–Ansatz, da dort ebenso aus einer sprachneutralen IDL-Schnittstellenbeschreibung mithilfe von Sprachbindungen eine Stellvertreterschnittstelle in der Verwendungssprache erzeugt wird.

Die Bindungen für eine Sprache assoziieren jeweils ein Konzept des Schnittstellenmetamodells mit einem korrespondierenden Konzept der Prozessmodellierungssprache. Um Bindungen auf der Ebene konkreter Prozessfragmente repräsentieren zu können, ist es jedoch erforderlich, die grundsätzliche Struktur von Bindungsinformationen bereits auf der Metameta-Ebene vorzugeben. Diese Struktur wird im Bindungs-Metametamodell (BM2-Modell) definiert und umfasst eine Datenbehälter-, Schnittstellen- und Aktivitätsbindung.

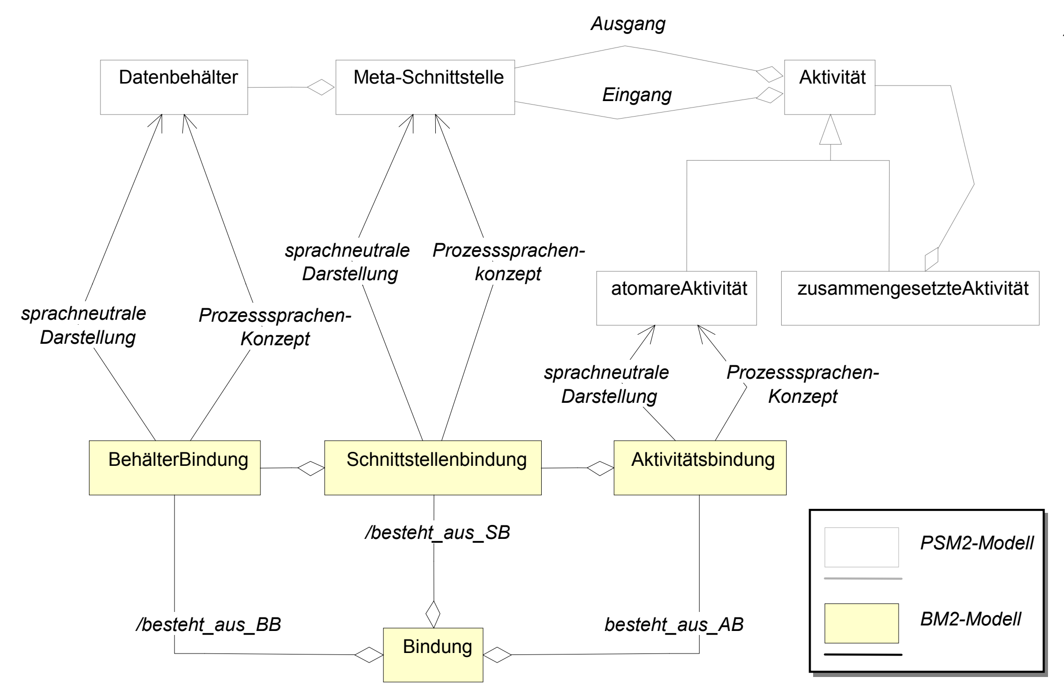


***Abb. 30**: Bindungs-M2-Modell*

Datenbehälterbindung

Für den Anschluss einer Kontextkomponente an den Datenfluss in einer übergeordneten PK-Komponente müssen deren Schnittstellenparameter durch entsprechende Konzepte der spezifischen Prozesssprache dargestellt werden. Dazu bindet die Instanz einer `Behälterbindung` das Komponentenkonzept `Situationsteil` an ein `Datenbehälter`-Konzept einer spezifischen Prozesssprache. Die Parameterbindung assoziiert daher zwei Instanzen der M2-Klasse `Datenbehälter`, eine für das Schnittstellenmetamodell und eine für die jeweilige Prozesssprache.

Schnittstellenbindung

Auf ähnliche Weise bindet eine Schnittstellenbindung das Komponentenkonzept einer `Schnittstelle` an ein entsprechendes `Schnittstellen`-Konzept der Prozesssprache. Mit der Instanziierung einer Schnittstellenbindung können Situationsschnittstellen in der Verwendungssprache abgebildet werden. Für die Abbildung müssen nicht nur das Konzept der Schnittstelle selbst, sondern auch die entsprechenden Datenbehälter in den verschiedenen Rollen der Schnittstelle abgebildet werden. Eine Schnittstellenbindung besteht daher aus mehreren `Behälterbindun`–

gen, die jeweils einen Situationsteil der Komponentenschnittstelle an das identifizierte Parameterkonzept der Prozesssprache binden.

Bei der Abbildung muss gewährleistet sein, dass die Komponentenschnittstelle vollständig an den Datenfluss in der PK-Komponente angeschlossen ist, d.h. dass (1) alle deklarierten Situationsteile tatsächlich an eine Datenbehälter-Instanz der Verwendungssprache gebunden werden und dass diese Abbildung (2) injektiv und (3) eindeutig ist. Dies kann bereits auf der Ebene des M2-Modells durch O-Telos-Metaregeln zugesichert werden, da alle integrierten Prozesssprachen ein Datenbehälter- und Schnittstellen-Konzept instanziieren und so die Metaregel auf deren Instanzen Bezug nehmen kann. Für die Formalisierung der dargestellten Integritätsbedingungen sind drei O-Telos-Zusicherungen erforderlich, von denen hier nur eine exemplarisch dargestellt wird; die formale Darstellung der beiden anderen Zusicherungen findet sich in [Schm99].

```
MetametaClass SchnittstellenBindung in Class with
  constraint
    alleGebunden    : $ vgl. [Schm99]     $
    einfacheBindung : $ vgl. [Schm99]     $
    injektiv        : $
      forall ssb,pb1,pb2,p1,p2,q1,q2 /VAR
             a1,a2,a3,a4,a5,a6,l1,l2,l3,l4,l5,l6 /VAR
             SSB /SchnittstellenBindung
             BB /BehälterBindung
             B /Datenbehälter
             PSB / BehälterBindung!prozesssprachenKonzept
             KKB / BehälterBindung!sprachneutraleDarstellung
             DB / SchnittstellenBindung!hat_behälterBindung
             ( P(a1,ssb,l1,pb1) and (a1 in DB)  and
               P(a2,ssb,l2,pb2) and (a2 in DB)  and
               P(a3,pb1,l3,p1)  and (a3 in KKB) and
               P(a4,pb2,l4,p2)  and (a4 in KKB) and
               P(a5,pb1,l5,q1)  and (a5 in PSB) and
               P(a6,pb2,l6,q2)  and (a6 in PSB) and
               (q1==q2) )
             ==>
             p1==p2 $
end
```

Aktivitätsbindung

Die Aktivitätsbindung entspricht der hat_Subkontext-Beziehung des NATURE-Prozessmodells und repräsentiert die Verwendung einer Kontextkomponente in einer PK-Komponente. Die Aktivitätsbindung bindet das Konzept einer Kontextkomponente an das atomare Aktivitätskonzept der Prozessmodellierungssprache. Für die vollständige Abbildung einer Komponente ist es erforderlich, dass die Komponente selbst sowie deren Ein- und Ausgangsschnittstelle in der Verwendungssprache dargestellt wird. Daher besteht die Aktivitätsbindung aus weiteren Subbindungen, die jeweils die Bindung der Ein- und Ausgangsschnittstelle der Kontextkomponente repräsentieren.

Bindung

Eine `Bindung` fasst alle Informationen zusammen, die zur Bindung einer Kontextkomponente an die Konzepte einer spezifischen Prozesssprache erforderlich sind. Dieses Konzept aggregiert demnach alle zusammengehörenden Aktivitäts-, Schnittstellen- und Datenbehälterbindungen.

Neben dem Aspekt der Formalisierung der Abbildungsregeln dienen Bindungen dazu, die notwendigen *Verwendungsinformationen* von Kontextkomponenten in einer PK-Komponente für einen Ausführungsmechanismus festzuhalten. Da die in der PK-Komponente dargestellte Komponentenschnittstelle nur ein Stellvertreter einer Komponente ist, die anderweitig implementiert wird, muss z.B. ein SLANG-Interpreter unterscheiden, ob es sich bei einer zu schaltenden Transition um eine reguläre Transition handelt, oder ob die Transition eine Komponente repräsentiert, die über ihre Stellvertreterschnittstelle einen Dienst anbietet. Ist dies der Fall, dann kann der Dienst über eine externe Prozessmaschine in Anspruch genommen werden, d.h. die Parameter werden an der Schnittstelle übergeben und die mit der Transition gebundene Komponente wird angesprochen.

6.4.2 Integrationsmethodik

Nach der Formalisierung der Sprachkonzepte der zu integrierenden Prozesssprachen sowie der Bindungskonzepte können wir aus dem M2-Modell nun eine Methodik für die Integration einer Prozesssprache mit dem Schnittstellenmodell für Kontextkomponenten ableiten. Das Ziel besteht in der Nutzung der Prozesssprache zur Spezifikation von Abläufen von Plankontexten.

6.4.2.1 Überblick

Abb. 31 gibt einen Überblick über die grundsätzliche Vorgehensweise bei der Integration einer Prozesssprache. Das Prozesssprachen-M2-Modell und das Bindungs-M2-Modell abstrahieren auf der obersten Ebene von den spezifischen Sprachkonzepten des Schnittstellenmetamodells, des Metamodells der zu integrierenden Prozesssprache und des Prozesssprachen-spezifischen Bindungsmodells. Die Konzepte des Schnittstellenmetamodells für Kontextkomponenten sind dabei bereits vorinstanziiert (Schritt 0; siehe auch Tab. 9 auf Seite 154). Für die Integration einer Prozesssprache müssen alle Konzepte des PSM2-Modells im Metamodell der Prozesssprache identifiziert und als PSM2-Konzepte instanziiert werden (Schritt 1). Die Integration der Prozesssprache mit dem Schnittstellenmetamodell erfolgt dann durch Instanziierung eines sprachspezifischen Bindungsmodells, das die Repräsentation einer Komponentenschnittstelle in der Prozesssprache ermöglicht (Schritt 2). In einem abschließenden Schritt werden zusätzliche Abbildungsregeln, die nicht schon durch die Struktur bzw. die Metaregeln des M2-Modells vorgegeben sind, sprachspezifisch auf der Ebene der Sprachkonzepte festgelegt (Schritt 3).

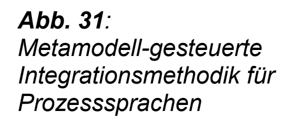
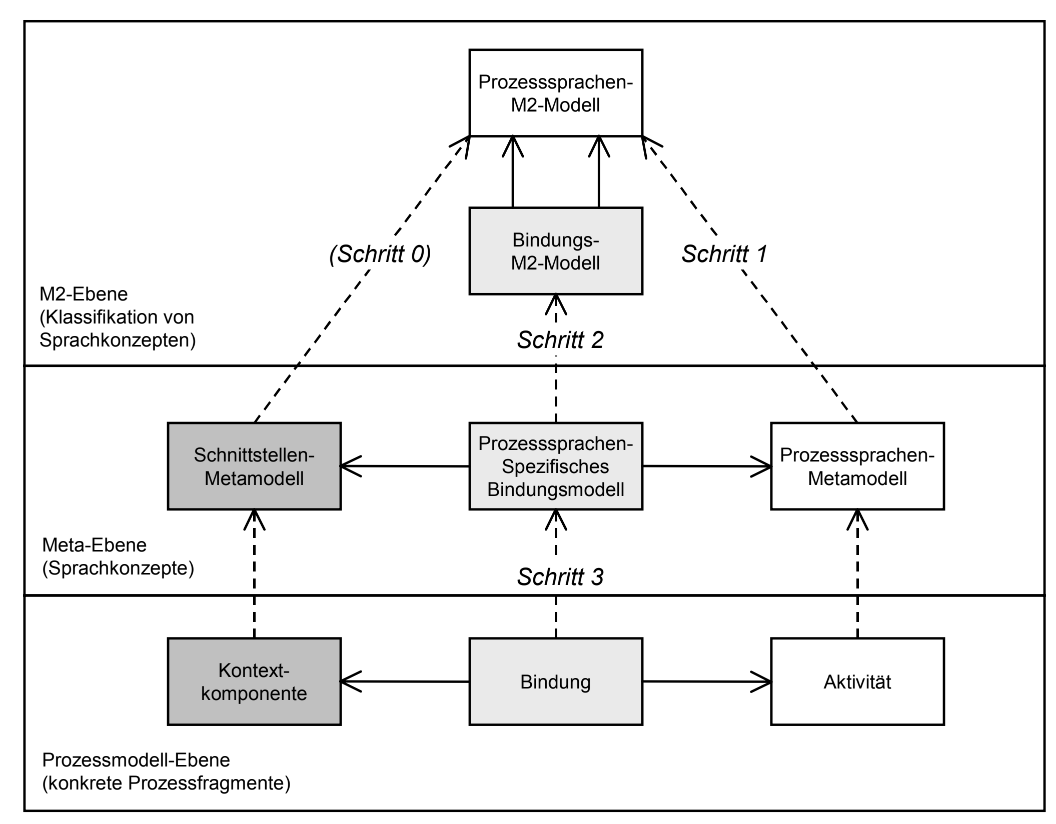


***Abb. 31**: Metamodell-gesteuerte Integrationsmethodik für Prozesssprachen*



6.4.2.2 Beispiel: Integration von SLANG-Netzen

Die Instanziierung des M2-Modells und die daraus abgeleiteten Teilschritte der Integrationsmethodik illustrieren wir nun am Beispiel der Einbettung des Petrinetz-Dialekts SLANG [BaFG93][21].

Schritt 1: Identifikation der M2-Konzepte im SLANG-Metamodell

Im ersten Schritt wird ein Metamodell der zu integrierenden Sprache SLANG erstellt und als Instanz des PSM2-Modells dargestellt. Für unsere Zwecke genügt dazu ein vereinfachtes Metamodell, welches fortgeschrittene Sprachmittel (z.B. die reflexiven Eigenschaften von SLANG zur Laufzeit-Modifikation von Aktivitätsnetzen) vernachlässigt (siehe Abb. 32).

Eine `SLANG-Aktivität` besteht aus `Stellen` und `Transitionen`, die über Kanten miteinander verbunden sind. Dabei werden die Eingangsstellen zum `Vorbereich` und Ausgangsstellen zum `Nachbereich` einer Transition zusammengefasst. Im Allgemeinen folgen `SLANG-Aktivitätsnetze` der Petrinetzsemantik, wobei die Sprache darüber hinaus ein `Wächter`- und ein `Aktions`-Konzept anbietet. Weiterhin können SLANG-Aktivitätsnetze in Subaktivitäten dekomponiert werden.

Gemäß Schritt 1 der oben skizzierten Integrationsmethodik können die von der Sprache SLANG bereitgestellten Konzepte wie in Abb. 32 dargestellt als PSM2-Konzepte instanziiert werden. Als zentrales Aktivitätskonzept finden wir

---

[21] Eine analoge Anwendung der Integrationsmethodik für UML-Zustandsdiagramme kann aus Platzgründen an dieser Stelle nicht eingehend dargestellt werden. Hier sei für eine detaillierte Darstellung auf [Schm99] verwiesen.

die SLANG-Aktivität, die in Form einer Transition entweder eine atomare Aktivität oder als Aktivitätsnetz eine zusammengesetzte Aktivität verkörpert. Über den (Vor- bzw. Nach-) Bereich einer SLANG-Aktivität lässt sich in SLANG ein Schnittstellenkonzept realisieren. Dabei übernehmen die (getypten) Stellen die Rolle von Datenbehältern. Da alle PSM2-Konzepte in der Sprache SLANG identifiziert werden können, kann die Sprache grundsätzlich mit dem Schnittstellenmodell integriert werden.

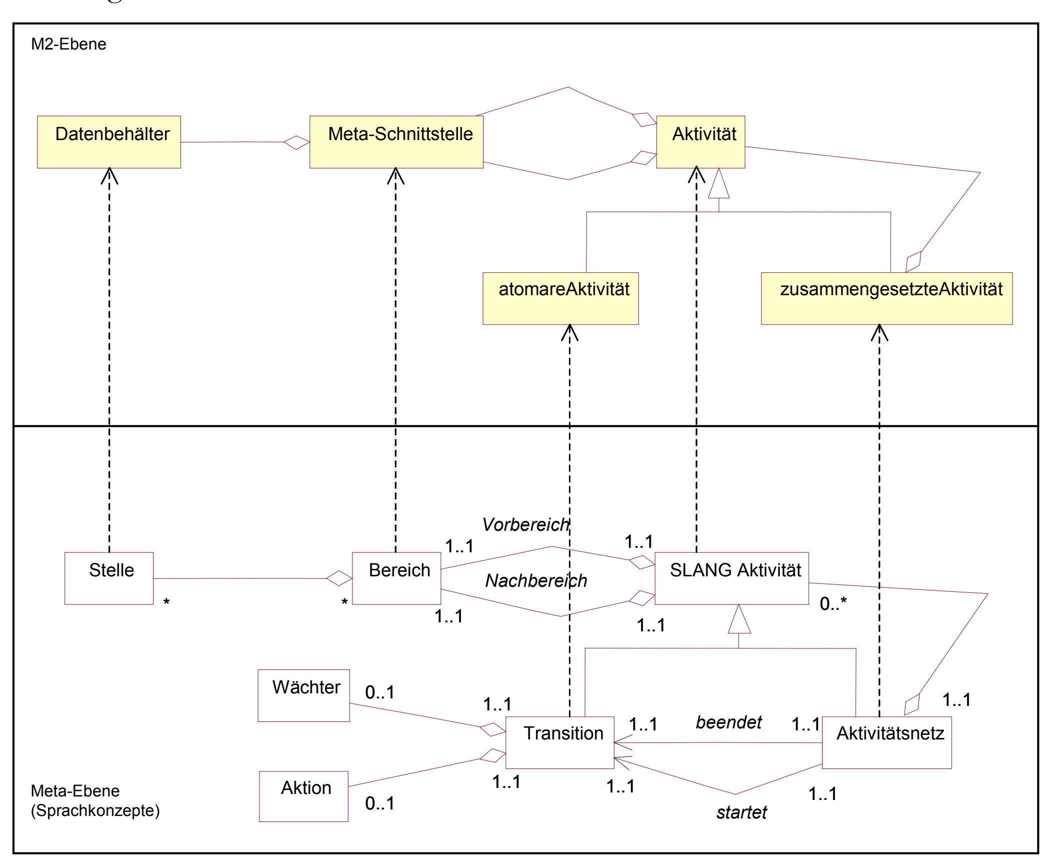


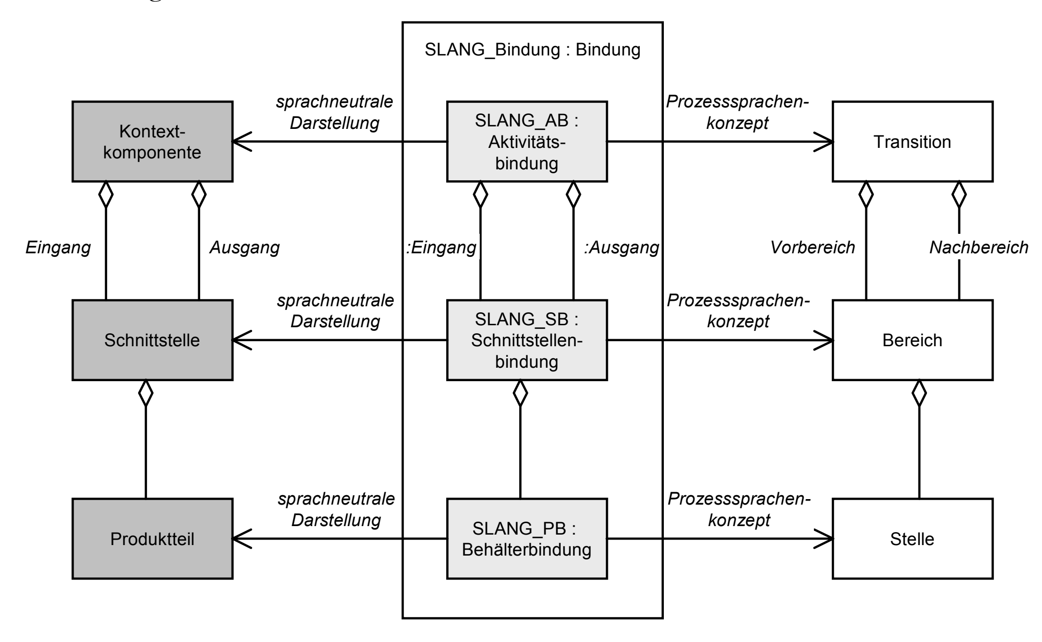
***Abb. 32**: Klassifikation des SLANG-Metamodells*

Schritt 2: Verwendungsbindungen für SLANG

Im zweiten Schritt werden die entsprechend dem PSM2-Modell klassifizierten SLANG-Konzepte an die Konzepte des Schnittstellenmetamodells gebunden (siehe Abb. 33). Die komplette Bindungsinformation für die Verwendung einer Kontextkomponente in SLANG wird durch die Klasse `SLANG_Bindung` repräsentiert. Instanzen dieser Klasse halten den Verwendungspunkt einer Kontextkomponente in einem SLANG-Netz fest. Die `SLANG_Bindung` besteht entsprechend der Struktur des Bindungs-M2-Modells aus einer `Aktivitätsbindung`, einer `Schnittstellenbindung` und einer `Behälterbindung`. Die Aktivitätsbindung `SLANG_AB` assoziiert eine Kontextkomponente mit einer Transition. Die Verwendung einer Kontextkomponente in einem SLANG-Aktivitätsnetz wird somit durch eine Transition repräsentiert. Um die Abbildung einer Komponentenschnittstelle in einem SLANG-Netz repräsentieren zu können, wird eine SLANG-Schnittstellenbindung (`SLANG_SB`) instanziiert. Dabei wird eine Ein- bzw. Ausgangsschnittstelle einer Kontextkomponente durch den Vor- bzw. Nachbereich einer Transition repräsentiert. Die Behälterbindung `SLANG_PB` drückt aus, dass ein Produktteil des Schnittstellenmetamodells durch eine Stelle in einem SLANG-Netz dargestellt wird. Die strukturell korrekte Zuordnung von Produktteilen auf Stellen ist bereits

durch die im vorigen Abschnitt formulierten Metaregeln auf der Ebene des M2-Modells zugesichert.

Abb. 33: SLANG-Bindung



Schritt 3: Sprachspezifische Abbildungsregeln

In Schritt 3 werden ergänzend zu den bereits durch die M2-Ebene vorgegebenen sprachneutralen Bindungsregeln, die eine korrekte Abbildung der Schnittstellenstruktur in der Verwendungssprache sicherstellen, zusätzliche sprachspezifische Abbildungsregeln hinzugefügt. Diese Regeln sind auf der mittleren, sprachspezifischen Ebene in Abb. 31 angesiedelt.

Im konkreten Fall der SLANG-Integration müssen wir die Typsicherheit der abgebildeten Schnittstellenparameter gewährleisten und die Abbildung von EK-Komponenten in einem SLANG-Netz festlegen.

Typisierung der Behälterbindung

Die Forderung nach Typsicherheit verlangt, dass ein im Schnittstellenmodell definiertes Produktteil und die Stelle, die diesen Produktteil in einem SLANG-Netz repräsentiert, den gleichen Typ haben. Dies wird durch die O-Telos-Zusicherung `typGleichheit` gewährleistet.

```
MetaClass SLANG_PB in BehälterBindung with
  Constraint
    typGleichheit :
      $ forall spb /SLANG_PB
               p / Produktteil
               tp, ts / Typ
               ( (spb sprachneutraleDarstellung p) and
                 (p hatTyp tp) and
                 (spb ProzesssprachenKonzept s) and
                 (s hatTyp ts) )
                ==> tp == ts $
end
```

Die gewählte Kontextalternative einer EK-Komponente muss in SLANG geeignet repräsentiert werden, damit der Kontrollfluss in Abhängigkeit von der gewählten Alternative verzweigen kann (siehe Diskussion in Abschnitt 6.3.2). Allgemein

hängt die Darstellung davon ab, ob in der Prozesssprache das Konzept eines Wächters vorhanden ist, mit dem Bedingungen ausgedrückt werden können. Ist dies der Fall, dann kann bei der Verwendung einer EK-Komponente an der EK-Ergebnisschnittstelle ein zusätzlicher Datenbehälter die gewählte Kontextalternative repräsentieren. Der Datenbehälter nimmt zur Laufzeit die Instanz eines Kontrolldatentyps auf, dessen Attribut die gewählte Alternative repräsentiert. Mit Hilfe des Wächters kann das Attribut überprüft werden und der Kontrollfluss je nach gewählter Alternative verzweigen.

Verfügt die Prozesssprache über kein Wächterkonzept, wie es z.B. in regulären Petrinetzen und auch in Funsoft-Netzen [DeGr93] der Fall ist, dann muss die gewählte Alternative auf andere Weise repräsentiert werden. Dies kann z.B. dadurch erfolgen, dass für jede Alternative eine eigene Stelle an der Ergebnisschnittstelle existiert, die entsprechend der Kontextwahl zur Laufzeit mit einer Marke gefüllt wird. Da hier je nach Prozesssprache mehrere Möglichkeiten denkbar sind, wurden diese Abbildungsregeln nicht auf der M2-Ebene festgelegt, sondern werden auf der sprachspezifischen Ebene durch erweiterte Integritätsbedingungen der instanziierten Aktivitätsbindung definiert.

Repräsentation der EK-Auswahl

Die Repräsentation der gewählten Alternative einer EK-Komponente erfolgt in SLANG durch eine zusätzliche Stelle im Nachbereich der Transition. Die Stelle nimmt zur Laufzeit eine Marke auf, deren Wert die gewählte Alternative darstellt und von einem speziellen Typ Kontrolldaten ist. Mit Hilfe eines Wächters kann dann eine bedingte Verzweigung über die Alternativen gebildet werden, der den Wert der Marke testet. Diese Abbildungsregel für EK-Komponenten wird durch die zusätzliche Integritätsbedingung ekErgebnis der Aktivitätsbindung sichergestellt

```
MetaClass Slang_AB in Aktivitätsbindung with
constraint
      ekErgebnis:
        $ forall sab / SLANG_AB
            kk / Kontextkomponente
            ek/ EK_Verhaltenskomponente
            t / Transition
            b /Bereich
           (sab ProzesssprachenKonzept t) and
           (t Nachbereich b) and
           (svb sprachneutraleDarstellung kk) and
           (kk hat_Verhaltenskomponente vk)
           ==> exists s /Stelle
           (b hatStelle s) and (s hatTyp Kontrolldaten) $
end
```

Weiterhin werden die Ergebnisschnittstellen einer EK-Komponente auf einen Nachbereich einer die EK-Komponente repräsentierenden Transition abgebildet. Wie bereits in Abschnitt 6.3.2 erläutert, unterstützen EK-Komponenten alle Ergebnisschnittstellen ihrer Kontextalternativen. Bei der Abbildung der Komponentenschnittstelle auf die Sprache SLANG ist zu berücksichtigen, dass SLANG-Transitionen lediglich *eine* Ergebnisschnittstelle haben, die durch die Stellen im Nachbereich definiert ist. Um die Verwendung von EK-Komponenten zu ermöglichen, werden daher alle Ergebnisschnittstellen auf denselben Nachbereich abgebildet. Der Nachbereich ist somit an mehrere Situationsschnittstellen gebunden, so dass je nach Auswahl des Benutzers das Ergebnis der gewählten Kontextalterna-

tive dem Nachbereich dynamisch zugewiesen wird. Für jede Alternativschnittstelle existiert eine Bindung an den Nachbereich der Transition, die festlegt, wie das Ergebnis der Kontextalternative auf die Stellen abgebildet wird. Wenn die EK-Komponente mehrere verschiedene Ergebnisschnittstellen hat, dann muss für jede Schnittstelle eine eigene Schnittstellenbindung existieren, die die Zuweisung zu den SLANG –Nachbereichsstellen vornimmt (Integritätsbedingung `ekSchnittstelle`).

```
MetaClass Slang_AB in Aktivitätsbindung with
  constraint
    ekSchnittstelle:
      $ forall sab /Slang_AB
               kk /KontextKomponente
               vk /EK_Verhaltenskomponente
               s1,s2 / Schnittstelle
               (kk verhalten vk) and
               (kk ausgang s1) and (kk ausgang s2) and (s1!=s2)
               ==> exists ssb1,ssb2 / Slang_SB
                          b / Bereich
               (ssb1 sprachneutraleDarstellung s1) and
               (ssb1 ProzesssprachenKonzept b) and
               (ssb2 sprachneutraleDarstellung s2) and
               (ssb2 ProzesssprachenKonzept b)$
end
```

6.5 Fazit

Ausgangspunkt dieses Kapitels war die Feststellung, dass das NATURE-Prozessmodell nicht direkt als ausführbare Prozessmodellierungssprache verwendet werden kann, da es keine geeigneten Konzepte zur Formalisierung von Situationen und zur Festlegung eines Kontrollflusses in Plankontexten anbietet. Gegenstand des Kapitels war daher insbesondere die Einbettung eines Kontrollmodells in das NATURE-Prozessmodell.

Um Flexibilität und Anpassbarkeit an domänenspezifische Bedürfnisse zu gewährleisten, haben wir das Ziel verfolgt, anders als in früheren Ansätzen nicht mehr nur einen Ablaufformalismus (z.B. Petrinetze oder prozedurale Sprachen) fest vorzugegeben oder einen neuen zu erfinden, sondern (innerhalb gewisser Grenzen) die Auswahl aus einer Palette unterschiedlicher Formalismen anzubieten, die für einen betrachteten Ablauf jeweils besonders geeignet sind. Hieraus ergab sich das Problem der *Interoperabilität* verschiedensprachlicher Prozessfragmente, für das in der Literatur im wesentlichen drei Lösungsansätze bekannt sind: Interoperabilität über eine vereinheitlichte Zwischensprache, über einen gemeinsamen Laufzeitzustand oder über definierte Schnittstellen und Delegation (Abschnitt 6.2).

Aufgrund der besseren Skalierbarkeit haben wir uns für einen komponentenbasierten Modellierungsansatz entschieden, bei dem das NATURE-Prozessmodell so erweitert wurde, dass Kontexte als Komponenten mit definierten Schnittstellen modelliert und in einer Verwendungssprache unabhängig von ihrer Implementierung wiederverwendet werden können. Das vorgestellte Schnittstellenmetamodell dient dabei als Komponentenbeschreibungssprache, die ähnlich einer IDL-Schnittstellensprache unabhängig von einer spezifischen Verwendungssprache ist. Aus Sicht der Verwendung ist die interne Struktur einer Komponente irrelevant, da sie

zu den Implementierungsdetails zuzuordnen ist und somit nach außen nicht in Erscheinung treten darf.

Die Interoperabilität des Schnittstellenmetamodells mit einer Verwendungssprache wird mit Hilfe des M2-Modells erzielt. Das M2-Modell abstrahiert von den Basiskonzepten, die erforderlich sind, damit eine Prozesssprache integriert werden kann und legt damit einen Integrationsrahmen fest. Bei der Integration ist die Existenz eines expliziten Modul- und Schnittstellenkonzept der Sprache entscheidend. Das M2-Modell stellt darüber hinaus Bindungskonzepte bereit, mit deren Hilfe Stellvertreterschnittstellen in Verwendungssprachen gebildet werden.

Als etwas problematisch gestaltete sich die Kapselung der Kontextalternativen eines Entscheidungskontexts, da auf der einen Seite der Dienst der Benutzerauswahl über die Komponentenschnittstelle zu kapseln ist, auf der anderen Seite jedoch die Ergebnisschnittstellen der Kontextalternativen aus pragmatischen Gründen zugänglich sein müssen. Dieser Einblick in die innere Struktur einer Komponente verletzt zwar das Geheimnisprinzip, ist jedoch nicht zu vermeiden.

Aus dem M2-Modell wurde eine Integrationsmethodik abgeleitet, die auf die Modellierungssprache SLANG und auf UML-Zustandsdiagramme angewandt wurde. Beide Sprachen eignen sich für eine komponentenbasierten Modellierungsansatz, da sie jeweils die M2-Konzepte Aktivität, Schnittstelle und Datenbehälter instanziieren. Damit wurde gezeigt, dass Kontextkomponenten sprachübergreifend verwendet werden können, so dass aus Modellierungssicht die Interoperabilität von verschiedensprachlichen Prozessfragmenten gewährleistet ist. In diesem Kapitel wurde die *statische* Interoperabilität von Prozessmodellierungssprachen in der Modellierungsdomäne betrachtet. *Dynamische* Aspekte der interoperablen Ausführung verschiedensprachlicher Prozessfragmente werden sind Gegenstand von Abschnitt 7.5, wenn wir uns mit der Integration der Interpreter in der Leitdomäne und dem Entwurf des Interpreterrahmenwerkes beschäftigen.

Teil 3

Umsetzung und Anwendungserfahrungen

Kapitel 7

Das PRIME-Rahmenwerk

In diesem Kapitel beschreiben wir das *PRIME[22]-Rahmenwerk* und die hinter seinem Design stehenden Entwurfsüberlegungen. Das PRIME-Rahmenwerk ist ein Architekturvorschlag für prozessintegrierte Umgebungen, der die in Kapitel 3 erhobenen Anforderungen an eine engere Integration zwischen den Prozessdomänen umsetzt und auf den in den Kapiteln 5 und 6 vorgestellten Modellierungsansätzen basiert.

PRIME ist generisch in dem Sinne, dass es von der Entwurfsdomäne einer Modellierungsumgebung und damit von den zugrunde liegenden Produktmodellen, den darauf operierenden, spezifischen Entwurfswerkzeugen und den zu unterstützenden Prozessfragmenten abstrahiert. Die PRIME-Architektur identifiziert die Hauptkomponenten einer prozessintegrierten Modellierungsumgebung, deren Beziehungen zueinander sowie deren interne Strukturierung in wiederverwendbare und in spezifische Teilsysteme. Abschnitt 7.1 gibt einen ersten Überblick über die Hauptkomponenten der PRIME-Architektur und deren Zusammenspiel.

Prozessmodellkonformes Verhalten der Werkzeuge, wie es in Abschnitt 3.3 gefordert wurde, setzt eine Synchronisation zwischen der Leitdomäne und der Durchführungsdomäne voraus. In Ergänzung zur statischen Architektursicht definieren wir daher ein Interaktionsprotokoll, das den dynamischen Abgleich der Prozesszustände zwischen den Prozessdomänen durch den Austausch von Nachrichten beschreibt (Abschnitt 7.2)

Realisiert wurde die PRIME-Architektur in Form eines objektorientierten *Implementierungs-Frameworks*, das die generischen Anteile einer prozessintegrierten Modellierungsumgebung als vorgefertigte Software-Komponenten bereitstellt und anwendungsspezifische Erweiterungen an wohldefinierten *Variationspunkten* erlaubt. Das Gesamt-Framework zerfällt dabei in ein Werkzeug-Framework und ein Prozessmaschinen-Framework. In Abschnitt 7.3 beleuchten wir das Werkzeug-Framework genauer. Abschnitt 7.4 beschäftigt sich mit der Weiterentwicklung des Werkzeug-Frameworks zu einem generischen Wrapper für externe Werkzeuge. In Abschnitt 7.5 beschreiben wir das Prozessmaschine-Framework. Abschnitt 7.6 fasst die wesentlichen Resultate dieses Kapitels zusammen.

[22] PRIME: Process-Integrated Modeling Environments

7.1 Die PRIME-Gesamtarchitektur

In diesem Abschnitt betrachten wir PRIME zunächst aus der Vogelperspektive. Einer Diskussion der zugrunde liegenden, generellen Entwurfsphilosophie (Abschnitt 7.1.1) folgt ein grober Überblick über die wesentlichen Architekturbausteine in der Durchführungs-, Leit- und Modellierungsdomäne (Abschnitt 7.1.2).

7.1.1 Framework-basierter Entwurfsansatz

Vor einer Diskussion der einzelnen Architekturkomponenten wollen wir zunächst kurz auf die generelle Entwurfsphilosophie eingehen, die dem PRIME-Ansatz zugrunde liegt. Wie in der Einleitung zu diesem Kapitel bereits angedeutet wurde, ist PRIME selbst noch nicht direkt als spezifische, prozessintegrierte Entwurfsumgebung verwendbar, sondern stellt vielmehr ein *generisches, objektorientiertes Framework* zur Verfügung, mit dessen Hilfe auf verhältnismäßig einfache und schnelle Weise prozessintegrierte Umgebungen für *unterschiedliche* Entwurfsdomänen entwickelt werden können.

Merkmale der Framework-basierten Softwareentwicklung

Allgemein fassen Frameworks das Wissen über Funktionsweisen und architekturelle Strukturen eines Anwendungsbereichs zusammen und kondensieren dieses in einer wieder- und weiterverwendbaren Form [Grif98]. Ein Framework stellt somit ein Anwendungsgerüst für die Entwicklung einer bestimmten Klasse von Softwaresystemen dar. Das Framework gibt die Architektur bereits in wesentlichen Zügen vor und schafft so einen Kollaborationsrahmen für existierende und noch zu entwickelnde Teilkomponenten. Die generischen Teile eines Frameworks liegen dabei häufig nicht nur in Form von Spezifikationen, sondern bereits als vorgefertigte Softwarebausteine vor.

Variationspunkte

Die Austausch- und Erweiterbarkeit von Teilkomponenten in Richtung einer konkreten Anwendung lässt sich in einem Framework durch so genannte *hot spots* kennzeichnen [Pree97b]. An solchen interessanten Variationspunkten im Framework werden häufig ähnliche, aber doch differenzierte Funktionen bzw. konkretisierende Anpassungen benötigt. Dabei bleibt jedoch grundsätzlich der vom Framework vorgegebene Architekturrahmen erhalten. Für den Anschluss spezifischer Anwendungsfunktionalität an den Variationspunkten werden häufig Entwurfsmuster [GHJV95; Pree97a; Srin99] eingesetzt wie zum Beispiel das *Factory*-Muster, das die Framework-gesteuerte Erzeugung neuer Komponenten, die zunächst nicht Bestandteil des Frameworks waren, erlaubt, oder das *Bridge*-Muster, das dem Framework einen einheitlichen Zugang zu unterschiedlich realisierten (Fremd-)Komponenten mit ähnlicher Funktionalität ermöglicht. Hierbei werden in der Regel abstrakte Klassen, die vom Framework vorgegeben sind, durch die Framework-Erweiterungen spezialisiert und mit spezifischer Funktionalität angereichert, wobei objektorientierte Mechanismen wie Polymorphie und die späte Bindung von Methodenaufrufen ausgenutzt werden.

Architektur-Wiederverwendung führt zu Kontrollinversion

Ein Framework zielt nicht nur auf reine Code-Wiederverwendung ab, sondern insbesondere auch auf Architektur-Wiederverwendung. Gerade aus der letztgenannten Art der Wiederverwendung resultieren nach Ansicht vieler Autoren die bedeutenderen Potenziale für eine Produktivitätssteigerung bei der Software-Entwicklung [Pree97b; GHJV95; FaSJ99]. Als wichtige Konsequenz aus der durch Frameworks unterstützten Architektur-Wiederverwendung tritt das Phänomen der

Kontrollinversion auf. Bei der konventionellen Wiederverwendung auf Basis von Komponenten-Bibliotheken erstellt der Applikationsentwickler ein Softwaregerüst, das existierende Komponenten aufruft. Bei der Framework-basierten Entwicklung schreibt der Applikationsentwickler dagegen Softwarebausteine, die an definierten Variationspunkten in das Framework „gesteckt" und von diesem *aufgerufen werden*. Da das Interaktionsmuster der Teilkomponenten bereits festgelegt ist, liegt die Verantwortung für Steuerung des Kontrollflusses nicht mehr beim Nutzer eines Frameworks, sondern beim Framework selbst. Dem Anwendungsentwickler werden somit sehr viele und oft schwierige Entwurfsentscheidungen vom Framework abgenommen.

Die PRIME-Architektur folgt einem Framework-orientierten Entwurfsansatz, d.h. wesentliche Teile der Architektur liegen als bereits realisiertes Grundgerüst vor, in welches für eine konkrete Ausprägung einer prozessintegrierten Umgebung mit verhältnismäßig geringem Aufwand nur noch spezifische Funktionsbausteine eingefügt werden müssen. Das Gesamt-Framework zerfällt dabei in zwei größere Sub-Frameworks. In der Durchführungsdomäne steht dem Entwickler ein Framework für prozessintegrierte Werkzeuge zur Verfügung. Wesentliches Merkmal dieses *GARPIT* (**G**eneric **AR**chitecture for **P**rocess-**I**ntegrated **T**ools) genannten Frameworks sind ein generischer Interpreter für die werkzeugrelevanten Anteile des in Kapitel 5 skizzierten Umgebungsmodells sowie eine Reihe von abstrakten Adapterklassen für den Anschluss an spezifische Produktmodelle, GUI-Implementationen und Kommunikationsmechanismen. Wir stellen GARPIT in Abschnitt 7.3 im Detail vor.

GARPIT: ein Sub-Framework für prozessintegrierte Werkzeugen

Das Gegenstück zu GARPIT auf Seiten der Leitdomäne ist das *GARPEM* (**G**eneric **AR**chitecture for **P**rocess **E**nactment **M**echanisms) genannte Prozessmaschinen-Framework. Dieses Framework stellt allgemein verwendbare Kernfunktionalität für die Ausführung von Prozessfragmenten bereit und definiert Schnittstellen für die Einbettung spezifischer Prozessspracheninterpreter gemäß dem in Kapitel 6 vorgestellten Modellierungsansatz zur Integration heterogener Prozessformalismen. Die Detailarchitektur und Realisierung von GARPEM wird in Abschnitt 7.5 beschrieben.

GARPEM: ein Sub-Framework für Prozessmaschinen

Beide Frameworks sind durch die gemeinsam genutzten Anteile des Umgebungsmodells sowie über das in Abschnitt 7.2 noch zu beschreibende Interaktionsprotokoll aufeinander abgestimmt und stützen sich konsequenterweise auf eine Reihe gemeinsam verwendeter Basiskomponenten ab. Ein wichtiges Kennzeichen des PRIME-Rahmenwerks ist, dass die komplette Integration zwischen den Prozessdomänen bereits auf Frameworkebene und durch die modellierungsseitige Verzahnung der Prozess- und Werkzeugmodelle im Umgebungsmodell vorweggenommen ist. Ein Umgebungsentwickler, der eine prozessintegrierte Entwicklungsumgebung auf Basis des PRIME-Rahmenwerks erstellt, braucht sich also um die architekturelle und technische Integration zwischen Modellierungs-, Durchführungs- und Leitdomäne nicht mehr zu kümmern.

Das Framework übernimmt die architekturelle und technische Integration der Prozessdomänen

7.1.2 Überblick über die Gesamtarchitektur

Abb. 34:
Grobarchitektur einer PRIME-basierten Modellierungsumgebung

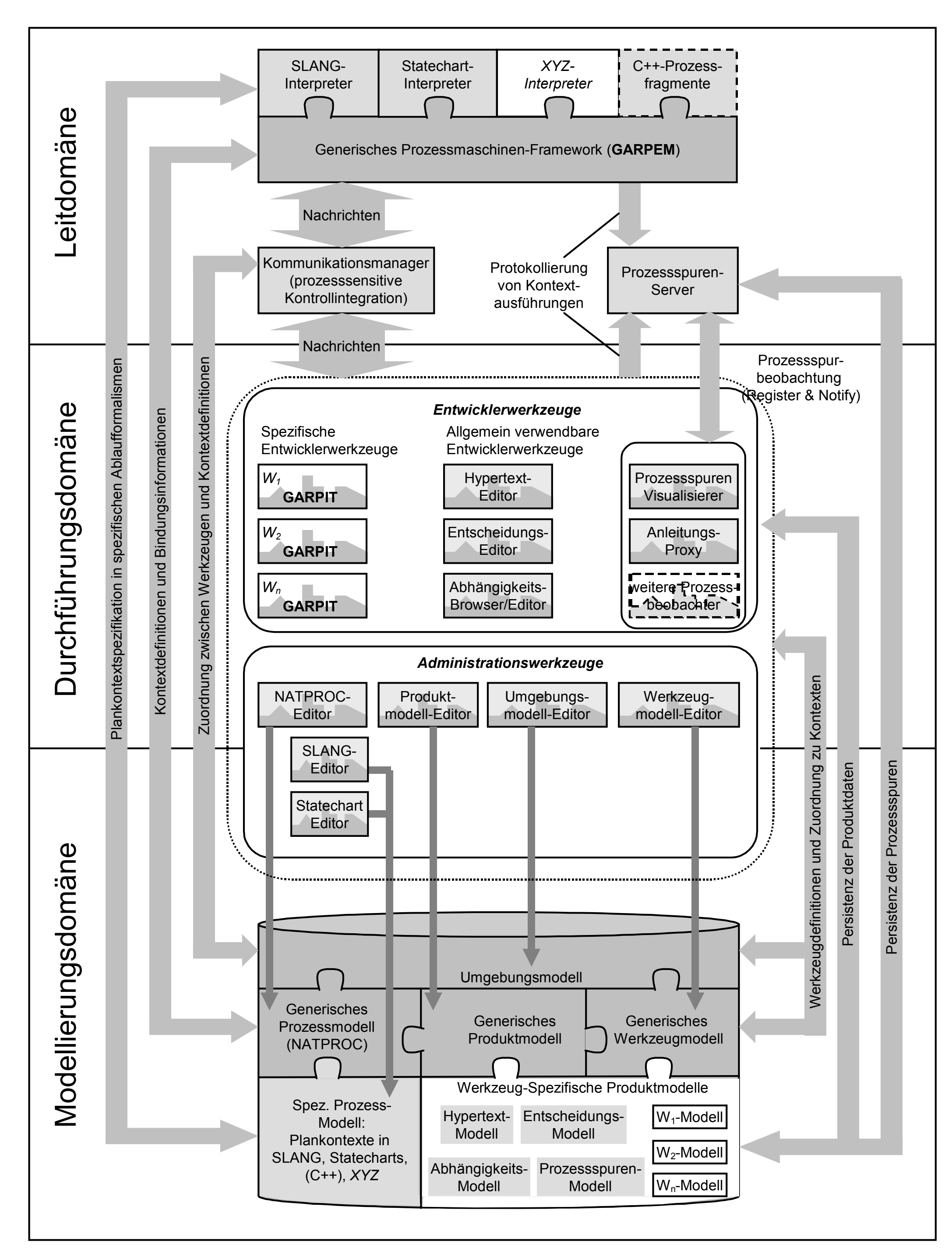


Abb. 34 vermittelt einen Überblick über die Gesamtarchitektur einer PRIME-basierten prozessintegrierten Modellierungsumgebung. Neben der Identifikation der Hauptkomponenten in den drei Prozessdomänen und ihrer Beziehungen untereinander liegt das Augenmerk dieser Überblicksdarstellung auf einer Unterscheidung des *Generizitätsgrades* der einzelnen Komponenten. Die reinen Framework-Anteile einer PRIME-basierten Umgebung sind dunkelgrau dargestellt. Diese Softwarekomponenten und Repository-Schemata stellen halbfertige Anwendungsgerüste für prozessintegrierte Entwicklungswerkzeuge bzw. Prozessmaschinen dar und müssen an den dafür vorgesehenen Variationspunkten um spezifische Funkti-

onalität ergänzt werden. PRIME bringt bereits eine Reihe von vorgefertigten Erweiterungen der Frameworkkomponenten zu lauffähigen und allgemein verwendbaren Anwendungen mit, die jeder PRIME-basierten Umgebung zur Verfügung stehen. Diese sind als hellgraue Architekturbausteine dargestellt. Die weiß dargestellten Architekturanteile markieren Platzhalter für spezifische Umgebungskomponenten, die in einer konkreten Modellierungsumgebung erst noch als Framework-Erweiterungen zu entwickeln bzw. zu integrieren sind.

7.1.2.1 Werkzeuge der Durchführungsdomäne

In der Durchführungsdomäne sind die *Entwicklerwerkzeuge* angesiedelt, deren Grundlage das GARPIT-Framework darstellt. Die wesentlichen Aufgaben des GARPIT-Frameworks (in Abb. 34 durch die dunkelgrauen Anteile der einzelnen Werkzeugsymbole repräsentiert) sind die Koordination der Interaktion mit der Prozessmaschine sowie die Interpretation und Ausführung der werkzeugrelevanten Anteile des Umgebungs- und Werkzeugmodells. Die Anwendung des GARPIT-Framework zur Entwicklung spezifischer Werkzeuge wird in Abb. 34 durch die Werkzeuge $W_1,...,W_n$ repräsentiert. Die interne Architektur des GARPIT-Frameworks und die Schnittstellen zur Anbindung spezifischer Werkzeugfunktionalität werden in Abschnitt 7.3 im Detail behandelt.

Als „Grundausstattung" sind in PRIME bereits eine Reihe voll funktionsfähiger, GARPIT-basierter Werkzeuge enthalten, die sich unabhängig vom konkreten Anwendungsbereich für unterschiedliche Modellierungsumgebungen als sehr nützlich erwiesen haben und daher allgemein verwendbar sind. Diese Werkzeuge sind ebenso wie jedes andere PRIME-Werkzeug innerhalb des Werkzeugmodells modelliert, erben vom GARPIT-Framework eine entsprechende Interpreterkomponente und die Schnittstelle zur Prozessmaschine und sind somit vollständig prozessintegrierbar. Im Folgenden skizzieren wir kurz die Funktionalität dieser Werkzeuge (siehe Abb. 35).

Hypertext-Editor

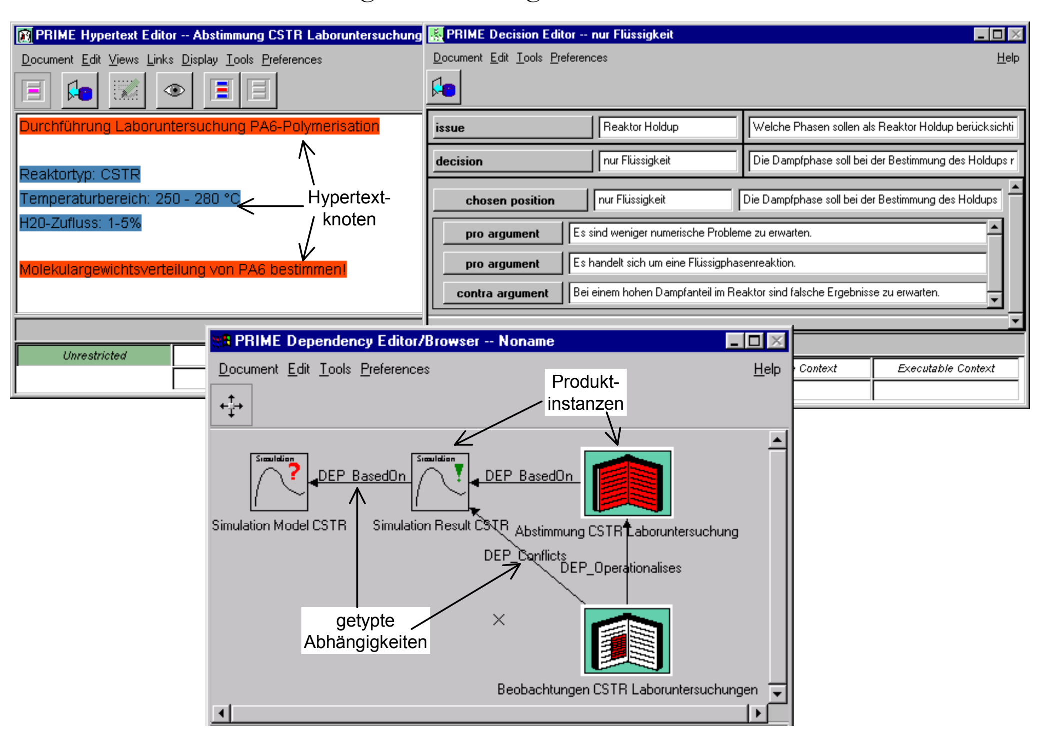
Für die Erfassung und Strukturierung von textuellen Dokumenten wie informal notierten Anforderungen und Spezifikationen, Sitzungsprotokollen oder Modell- und Softwaredokumentationen steht in PRIME ein *Hypertext-Editor* zur Verfügung (siehe Abb. 35, oben links). Hypertexte lassen sich in einzelne Textknoten zerlegen, die disjunkte Teilstücke des Textes beinhalten. Gruppen von (nicht notwendigerweise zusammenhängenden) Hypertext-Knoten können zu *Hypertext-Sichten* zusammengefasst werden. Der Hypertext-Editor stellt grundlegende Funktionalitäten zur Edierung von Hypertext-Dokumenten und deren Strukturierung in Knoten und Sichten bereit. Innerhalb des dem Hypertext-Editor zugrunde liegenden Produktmodells können komplette Hypertext-Dokumente ebenso wie Sichten oder einzelne Knoten als individuelle Produkte referenziert werden. Einzelheiten zu den formalen Grundlagen des verwendeten Hypertextmodells sowie zum Hypertext-Editor selbst können [PoHa95; Pohl96] entnommen werden.

Entscheidungseditor

Der Entscheidungseditor dient der strukturierten Erfassung und Darstellung von Entwurfsentscheidungen und Begründungshistorien (siehe Abb. 35, oben rechts).

Angelehnt an das IBIS-Modell (Issue Based Information Systems [CoBe88]), lassen sich im Entscheidungseditor einer *Problemstellung* (issue) eine oder mehrere *Lösungsvorschläge* (positions) zuordnen, die mithilfe von *Pro-* und *Kontra-Argumenten* diskutiert werden. Durch Auswahl eines Lösungsvorschlags wird die Problemstellung einer *Entscheidung* (decision) zugeführt. Da sich die Sach- oder Erkenntnislage im Laufe des Entwicklungsprozesses ändern kann, erlaubt der Entscheidungseditor die nachträgliche Modifikation und das Hinzufügen von Lösungsvorschlägen und Argumenten, was gegebenenfalls zur Revision einer einmal getroffenen Entscheidung führt. Wie beim Hypertexteditor können auch die Teilkomponenten einer Entscheidung (Problemstellung, Lösungsvorschläge, Argumente) im Produktmodell des Entscheidungseditors feingranular referenziert werden.

***Abb. 35**:*
Die prozessintegrierten PRIME-Werkzeuge Hypertext-Editor, Entscheidungs-Editor und Abhängigkeits-Editor



Abhängigkeits-Editor

Mithilfe des grafischen Abhängigkeitseditor können getypte, feingranulare Abhängigkeiten zwischen beliebigen Produktinstanzen erzeugt und visualisiert werden (siehe Abb. 35, unten). In der grafischen Ansicht des Abhängigkeitseditors werden die einzelnen Produkte durch produkttypspezifische Symbole angezeigt, die durch gerichtete Abhängigkeitskanten miteinander verbunden sind. Mithilfe einer Browsing-Funktion kann der Benutzer ausgehend von einem ausgewählten Produkt alle abhängigen Produkte ermitteln und so in einem Netz von Abhängigkeiten entlang navigieren. Abhängigkeiten können entweder manuell durch den Benutzer oder während des Entwicklungsprozesses automatisch bei der Ausführung entsprechender Prozessfragmente angelegt werden.

7.1.2.2 Prozessmaschine

Laufzeit-Umgebung für die Ausführung verschiedensprachlicher Prozessfragmente

In der Leitdomäne koordiniert die Prozessmaschine die Ausführung von Plankontexten. Analog zu den Werkzeugen wird ein großer Teil der administrativen Basisfunktionalität durch das generische Prozessmaschinen-Framework GARPEM

bereit gestellt. Insbesondere realisiert das Framework das in Abschnitt 7.2 beschriebene Interaktionsprotokoll zwischen den Prozessdomänen aus Sicht der Leitdomäne. Weiterhin liefert GARPEM die koordinierende Laufzeit-Umgebung für die Ausführung geschachtelter, verschiedensprachlich definierter Plankontexten. Die Anbindung der dafür erforderlichen sprachspezifischen Interpreter wird in Abschnitt 7.4 im Detail beschrieben; bis jetzt wurden auf Basis des GARPEM-Frameworks Plankontext-Interpreter für die Formalismen SLANG und UML-Statecharts erstellt und integriert. Darüber hinaus ist es bereits seit der ersten Version des PRIME-Rahmenwerks möglich, Plankontexte als C++-Methoden zu definieren und in kompilierter Form direkt zum GARPEM-Framework zu binden.

7.1.2.3 Kommunikationsmanager

Wie bereits angedeutet, wird das Interaktionsprotokoll zwischen der Prozessmaschine und den Werkzeugen vollständig von generischen Komponenten der jeweiligen Frameworks realisiert. Bei der Interaktion zwischen den Prozessdomänen werden wechselseitig Kontextanforderungen und -resultate in Form von Nachrichten ausgetauscht. Dabei interagieren Prozessmaschine und Werkzeuge aus den in Abschnitt 3.3.3 diskutierten Gründen nicht in direkten Punkt-zu-Punkt-Beziehungen miteinander, sondern über einen Kommunikationsmanager, der die Kommunikationspartner voneinander abschottet. Der Kommunikationsmanager entspricht einer Message-Server-Komponente in Message-Broadcasting-Architekturen (siehe auch Abschnitt 3.3.3), sorgt jedoch im Gegensatz zu Ansätzen wie Field [Reis90; Reis90a], Softbench [Caga90] oder ToolTalk [Fran91; Suns93] für eine *prozessmodellkonforme* Verteilung der Kontextnachrichten, d.h. die Auslieferung von Kontextnachrichten wird entsprechend der Zuordnung zwischen Kontexten und Werkzeugkategorien bzw. Prozessmaschine im Umgebungsmodell vorgenommen.

Prozesskonforme Verteilung von Kontextnachrichten

Zur Laufzeit führt der Kommunikationsmanager Buch über die aktuell laufenden Werkzeuginstanzen und die dort geladenen Produkte. Bei Bedarf startet er eine neue Werkzeug- oder Prozessmaschineninstanz. Die Kapselung des Wissens über die aktuell aktiven Werkzeug- und Prozessmaschineninstanzen im Kommunikationsmanager reduziert die Komplexität der Nachrichtenschnittstelle auf Seiten der Werkzeuge und der Prozessmaschine beträchtlich. Auf die vom Kommunikationsmanager ermöglichte Orts-[23] und Namenstransparenz werden wir bei der Darstellung des Interaktionsprotokolls zwischen Leit- und Durchführungsdomäne noch genauer eingehen (Abschnitt 7.2).

Orts- und Namenstransparenz

Über seine eigentliche Aufgabe als Nachrichtenverteiler hinaus verfügt der Kommunikationsmanager über eine Benutzerschnittstelle, über die der Benutzer Werkzeuge manuell starten kann.

7.1.2.4 Prozessspuren-Server

Protokollierung von Prozessspuren

Eine prozessintegrierte Umgebung sollte neben einer Anleitung durch präskriptiv modellierte Prozessfragmente auch die Nachvollziehbarkeit abgelaufener Prozesse

[23] Die Werkzeug- und Prozessmaschineninstanzen benötigen initial nur eine Referenz auf den Kommunikationsmanager, an den sie sich beim Start binden.

und die Akkumulierung und Nutzung von Erfahrungswissen unterstützen [JPRS94]. Grundvoraussetzung hierfür ist die persistente Protokollierung der in den Entwicklungswerkzeugen durchgeführten Ausführungs- und Entscheidungskontexte. Aus der Gesamtheit der protokollierten Kontextausführungen und der dabei involvierten Produktinstanzen ergibt sich die so genannte *Prozessspur*, anhand derer abgelaufene Prozesse zurück verfolgt werden können.

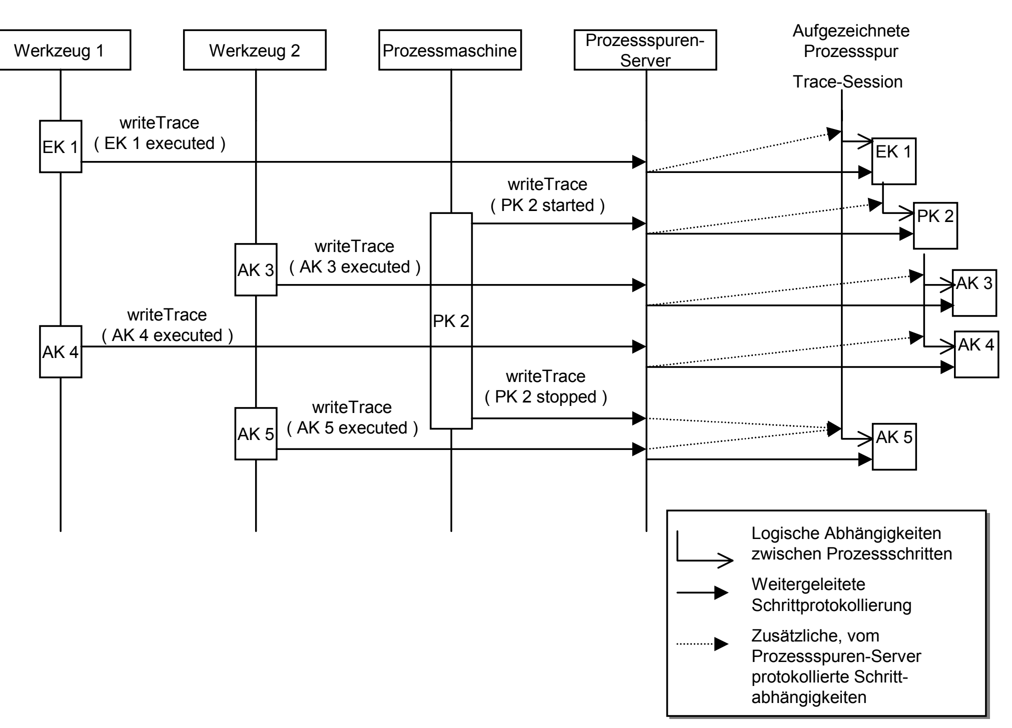
Aktive Rolle der Werkzeuge bei der Prozessspuraufzeichnung

Eine grundlegende Entwurfsentscheidung des PRIME-Frameworks besteht darin, dass die persistente Auszeichnung von Ausführungs- und Entscheidungskontexten jeweils von den Werkzeugen, in denen sie durchgeführt wurden, selbstständig und automatisch durchgeführt wird. Dies entlastet erstens den Entwickler von einer arbeitsaufwändigen und fehlerträchtigen manuellen Protokollierung seiner Arbeitsprozesse. Zweitens „kennen" die jeweiligen Werkzeuge die Semantik der aufgezeichneten Arbeits- und Entscheidungsschritte am besten, so dass hier qualitativ hochwertigere Nachvollziehbarkeitsinformationen gewonnen werden können als bei indirekten Beobachtungsansätzen, die wie etwa der in Abschnitt 3.3.5 beschriebene Provence-Ansatz zwangsläufig auf einer systemtechnisch tieferen Ebene ansetzen müssen.

Abhängigkeiten zwischen Teilschritten einer Prozessspur

Um eine semantisch reichhaltige Prozessspur zu erhalten, reicht es jedoch nicht aus, die einzelnen Arbeits- und Entscheidungsschritte isoliert voneinander von den Werkzeugen aufzeichnen zu lassen. Vielmehr müssen auch die *zeitlichen und logischen Abhängigkeiten* zwischen den einzelnen Schritten erfasst werden. Hier ergibt sich das Problem, dass die Prozessdurchführung über die unterschiedlichen Werkzeuge der Durchführungsdomäne verteilt stattfindet und kein einzelnes Werkzeug über eine globale Sicht auf die Abhängigkeiten zwischen den einzelnen Kontextausführungen verfügt. Auch die Prozessmaschine kommt hier als koordinierende Komponente für die Prozessspurenaufzeichnung nicht in Betracht, da sie nur während der Ausführung eines Plankontexts aktiv ist.

Abb. 36: Arbeitsweise des Prozessspuren-Servers



Stattdessen zentralisiert der *Prozessspuren-Server* den Dienst der Prozessspurenprotokollierung. Er nimmt von den Werkzeugen und der Prozessmaschine voneinan-

der unabhängige Informationen über die Anforderung und Ausführung von Kontexten entgegen, bringt diese in einen zeitlichen und logischen Zusammenhang und persistiert die Ausführungsinformationen samt Schrittabhängigkeiten im Prozess-Repository. Abb. 36 illustriert die Arbeitsweise des Prozessspuren-Servers, die der eines Transaktionsmonitors für die Bündelung von Datenbankenanfragen ähnelt. Einzelheiten zur Realisierung des Prozessspuren-Server und zum zugrunde liegenden Prozessspuren-Datenmodell werden zur Zeit in einer Diplomarbeit umfassend aufgearbeitet [Bran01].

7.1.2.5 Prozessbeobachter-Klienten

CORBA-basierter Notifikationsdienst für Prozessbeobachter-Klienten

Neben der reinen Protokollierung von Prozessspuren bietet der Prozessspuren-Server auch einen Notifikationsdienst an, über den sich so genannte Prozessbeobachter-Klienten über das Auftreten aller oder auch nur bestimmter Prozessspur-Ereignisse informieren lassen können (siehe auch Abb. 34). Dieser Notifikationsdienst ist gemäß dem Observer-Muster [GHJV95] realisiert, nutzt CORBA als Kommunikationsplattform und erlaubt daher eine netzwerkweite Verteilung von Prozessspuren-Server und -Klienten. Zurzeit wird der Notifikationsdienst von zwei generischen Werkzeugen des PRIME-Frameworks verwendet: dem *Prozessspuren-Visualisierer* und dem *generischen Anleitungswerkzeug*.

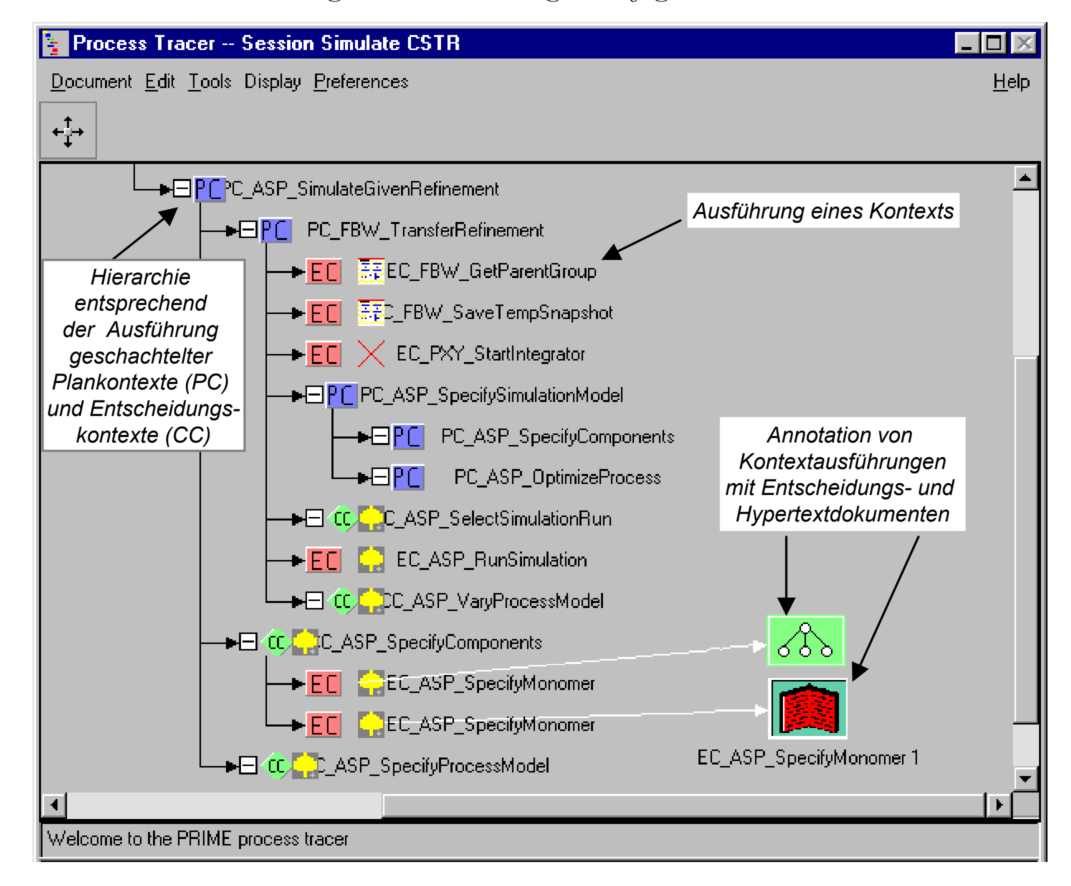


***Abb. 37**: Der prozessintegrierte Prozessspuren-Visualisierer*

Prozessspuren-Visualisierer

Der *Prozessspuren-Visualisierer* dient der hierarchischen Darstellung von Prozessspuren (siehe Abb. 37). Der Prozessspuren-Server leitet jede protokollierte Kontextausführung an den Prozessspuren-Visualisierer weiter, der die grafische Darstel-

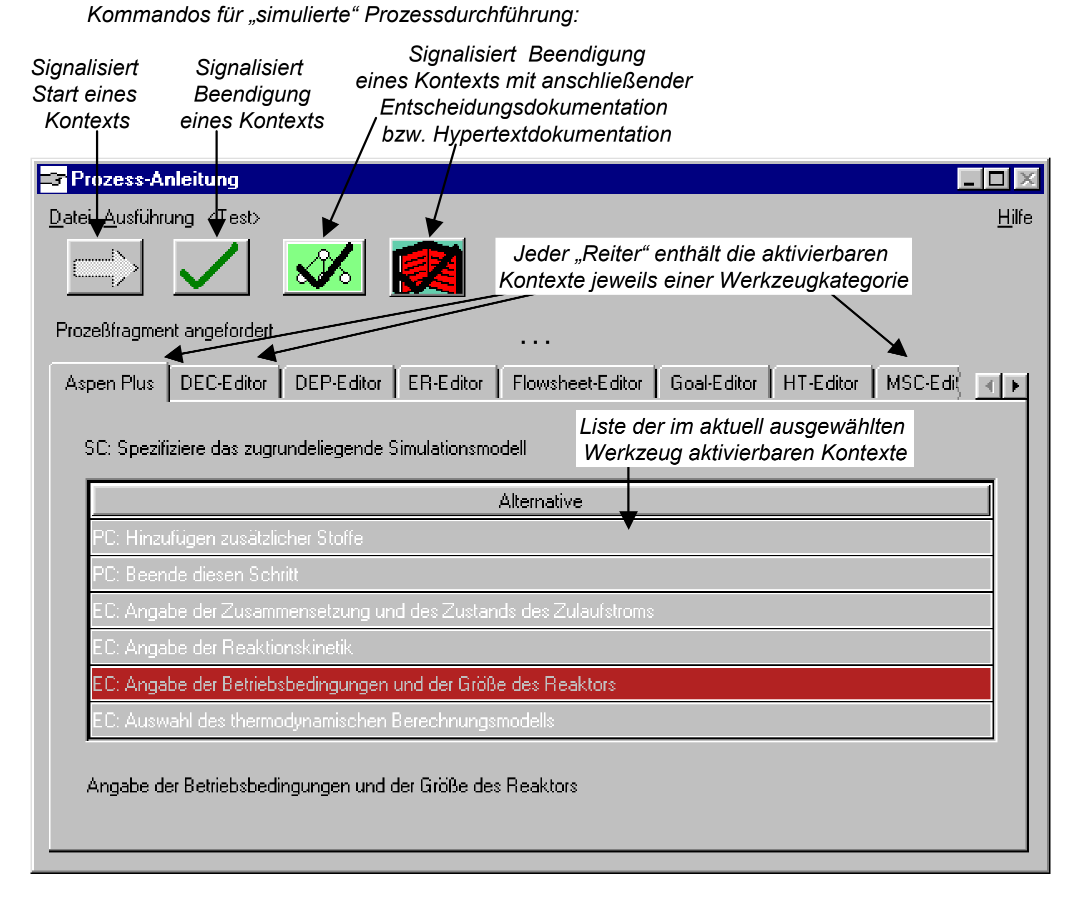
lung dynamisch fortschreibt. Daneben kann der Prozessspuren-Visualisierer auch gespeicherte Prozessspuren früherer Abläufe zur Unterstützung der Nachvollziehbarkeit darstellen. Über entsprechende Prozessfragmente ist der Prozessspuren-Visualisierer mit dem Hypertext-, Entscheidungs- und Abhängigkeitseditor integriert und erlaubt so die Annotation beliebiger Teilschritte oder -sequenzen einer Prozessspur mit informellen Texten, Entscheidungsdokumentationen oder beliebigen anderen Produkten einer PRIME-basierten Umgebung.

Anleitungswerkzeug

Alternative Schnittstelle zur Kontextaktivierung

Das generische Anleitungswerkzeug liefert eine alternative Benutzerschnittstelle für die Aktivierung von Werkzeugkontexten und die Entgegennahme von Rückmeldungen (Abb. 38). Sein Haupteinsatzgebiet liegt in der Einbindung externer Werkzeuge, die über keine geeigneten Möglichkeiten zur Anpassung der Benutzerschnittstelle an den aktuellen Prozesszustand und zur Rückmeldung der in diesen Werkzeugen durchgeführten Aktionen verfügen (siehe Abschnitt 7.4). In diesem Fall fungiert das Anleitungswerkzeug als *Proxy*, der gegenüber der PRIME-Umgebung die Rolle des externen Werkzeugs einnimmt.

***Abb. 38**: Das generische Anleitungswerkzeug*



Das generische Anleitungswerkzeug wurde so entworfen, dass es nicht nur externe Werkzeug nachbildet, sondern für alle Werkzeuge, also insbesondere auch die originären PRIME-Werkzeuge, eine alternative Schnittstelle zur Kontextaktivierung und -rückmeldung bereitstellt. In der grafischen Benutzerschnittstelle existiert für jede Werkzeugkategorie ein Teildialog („Reiter“), in der alle aktuell ausführbaren Kontexte dieses Werkzeugs aufgelistet sind. Um das Verhalten originärer PRIME-Werkzeuge dynamisch nachbilden zu können, muss das Anleitungswerk-

zeug die dort durchgeführten Aktion nachziehen. Zu diesem Zweck registriert sich das Anleitungswerkzeug beim Prozessspuren-Server und wird so über die in den PRIME-Werkzeugen durchgeführten Abläufe auf dem Laufenden gehalten.

7.1.2.6 Administrations- und Metamodellierungswerkzeuge

Zur Erstellung und Pflege der im Prozess-Repository abgelegten Prozess-, Werkzeug- und Produktmodelle existieren eine Reihe von Administrations- und Metamodellierungswerkzeugen. Wie in Abb. 34 angedeutet, nehmen diese Werkzeuge eine interessante Zwitterrolle zwischen der Modellierungs- und der Durchführungsumgebung ein. Konzeptuell können sie der Modellierungsdomäne zugeordnet werden, da sie primär den Methodeningenieur bei der Definition und Konfiguration der Prozess- und Werkzeugmodelle unterstützen. Aus einer technischen Perspektive betrachtet basieren die Administrations- und Metamodellierungswerkzeuge jedoch genau wie alle anderen Entwicklerwerkzeuge der Durchführungsdomäne auf dem GARPIT-Framework und sind daher vollständig prozessintegrierbar.

Nahtlose Verschränkung von Prozessausführung und -definition

Somit wird der Prozess der Prozessmodelldefinition durch die Definition geeigneter Meta-Prozessfragmente selbst wieder zum Gegenstand einer prozessintegrierten Werkzeugunterstützung. Daher profitiert nicht nur der Applikationsentwickler, sondern auch der Methodeningenieur von der Assistenzfunktion der Prozessintegration in einer PRIME-basierten Umgebung. Darüber hinaus ergeben sich interessante Möglichkeiten zur *prozessmodellgesteuerten Verschränkung* von Prozessausführung und -definition. Auf Basis entsprechender Prozessmodelle könnte beispielsweise der Applikationsentwickler bei der *ad-hoc-Definition* neuer Prozessfragmente mithilfe der Prozessmodellierungswerkzeuge und der anschließenden Einbringung dieser Prozessfragmente in die *laufenden* Entwicklungswerkzeuge angeleitet werden. In den derzeit existierenden Anwendungen des PRIME-Rahmenwerks haben wir dieses Potenzial zur Verschränkung von Entwicklungs- und Metaprozess erst in wenigen ausgewählten Nutzungsszenarien erprobt (siehe dazu auch Kapitel 8). Im Kontext des zurzeit viel diskutierten Problems der *Evolution laufender Prozesse* eröffnen sich hier jedoch interessante Anknüpfungspunkte für weitere Forschungsarbeiten.

Die Administrations- und Metamodellierungswerkzeuge sind datenseitig durch die in Kapitel 5 und 6 dargestellten Querbezüge zwischen den einzelnen Metamodellen (Prozessmodell, spezifische Plankontextmodelle, Werkzeugmodell, Umgebungsmodell, Produktmodell) miteinander integriert. Abb. 34 illustriert dies durch eine verzahnte Darstellung der entsprechenden Teilmodelle innerhalb des Prozess-Repositories. Die Definition eines neuen Prozessfragments und dessen Zuordnung zu einer Werkzeugkategorie involviert daher i.a. mehrere der im Folgenden vorgestellten Werkzeuge. Die dabei auftretenden werkzeugübergreifenden Abläufe werden durch entsprechende Prozessfragmente angeleitet.

NATPROC-Editor

Der NATPROC-Editor unterstützt die Verwaltung von Kontextkomponenten und deren Schnittstellen entsprechend den in Kapitel 5 und 6 vorgestellten Metamodellen.

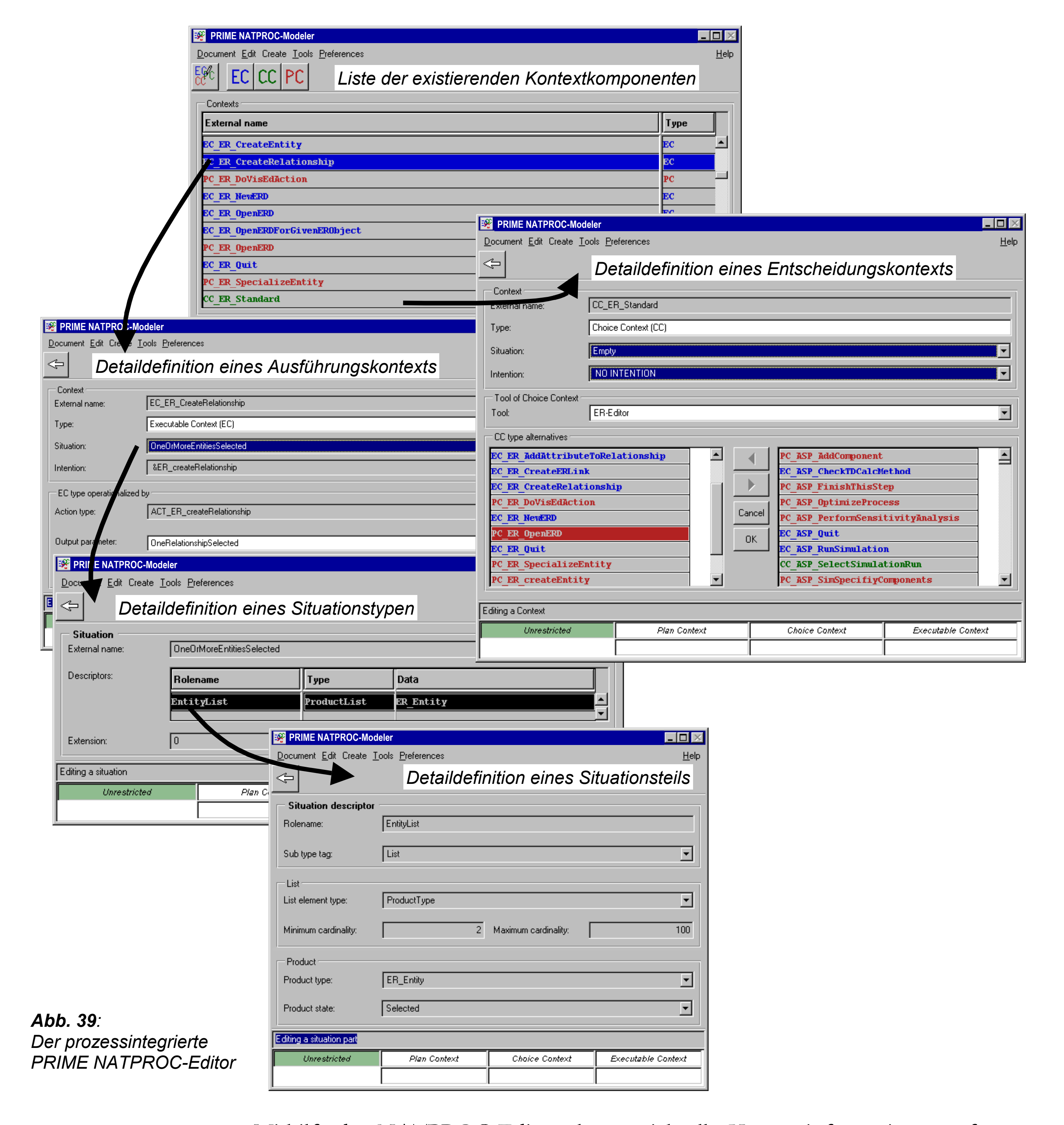


***Abb. 39**:*
Der prozessintegrierte PRIME NATPROC-Editor

Mithilfe des NATPROC-Editors lassen sich alle Kontextinformationen erfassen, die unabhängig von einem konkreten Formalismus zur Plankontextdefinition sind. Seine Grundfunktionalitäten bestehen demnach in der

- Definition von Intention;
- Definition von Situationen;
- Kombination von Intentionen und Situationen zu Kontexten;
- Definition von Aktionen und Zuordnung zu Ausführungskontexten;
- Zuordnung von Alternativen zu Entscheidungskontexten.

Darüber hinaus implementiert der NATPROC-Editor eine Reihe von Integritätschecks, mit deren Hilfe die Konsistenz der im Prozess-Repository abgelegten Kontextmodelle nachgeprüft werden kann, sofern dies nicht bereits datenbankseitig gewährleistet werden kann. Abb. 39 zeigt die unterschiedlichen Auswahl- und Eingabefenster des NATPROC-Editors und vermittelt einen Eindruck von der Funktionalität des Werkzeugs.

Plankontext-Editoren

Wie in Kapitel 6 beschrieben, erfolgt die Definition der Ablaufreihenfolge zwischen den Teilkontexten eines Plankontexts mithilfe spezifischer Ablaufformalismen. Hierzu stehen für die bislang in das Kontextmodell integrierten Formalismen (SLANG-Netze und UML-Statecharts) spezielle Editoren zur Verfügung, mit denen der NATPROC-Editor bei der Definition oder Modifikation von Plankontexten interagiert (Abb. 40).

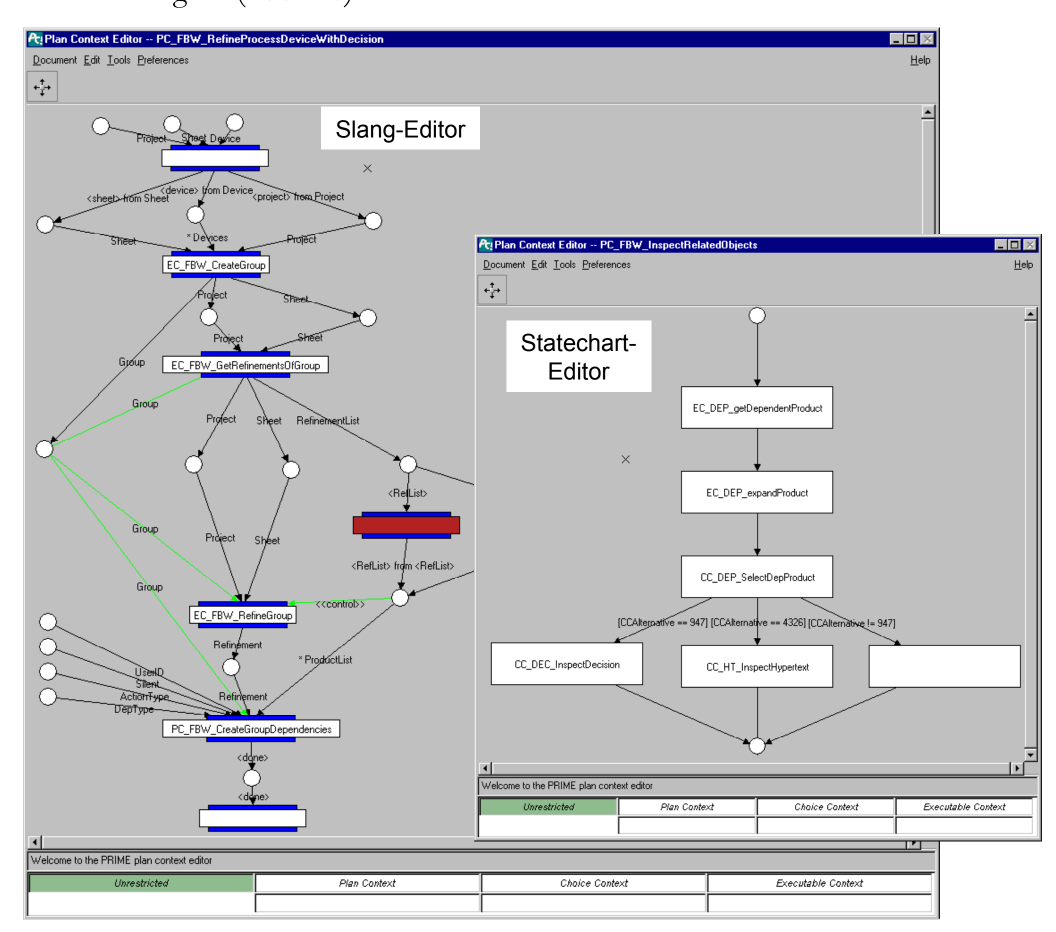


***Abb. 40**: Die prozessintegrierten Plankontext-Editoren (links SLANG, rechts UML Statecharts)*

Die Plankontext-Editoren bieten die folgenden Grundfunktionalitäten an:

- ❑ Grafische Edier- und Anzeigefunktionen für die Sprachelemente der jeweiligen Formalismen;
- ❑ Auswahl und Import von Kontextkomponenten
  - ❍ Erzeugung von Stellvertreterschnittstellen in der jeweiligen Sprache gemäß dem Bindungsmetamodell aus Abschnitt 6.4;
  - ❍ Erzeugung von Bindungsinformationen im Prozess-Repository;

- Ablaufvisualisierung bei der Ausführung von Plankontexten
  - SLANG-Editor: Belegung der Stellen mit Marken, Schalten von Transitionen;
  - Statechart-Editor: Aktiver Zustand, Schalten von Transitionen;
- Sprachspezifische Konsistenzüberprüfungen.

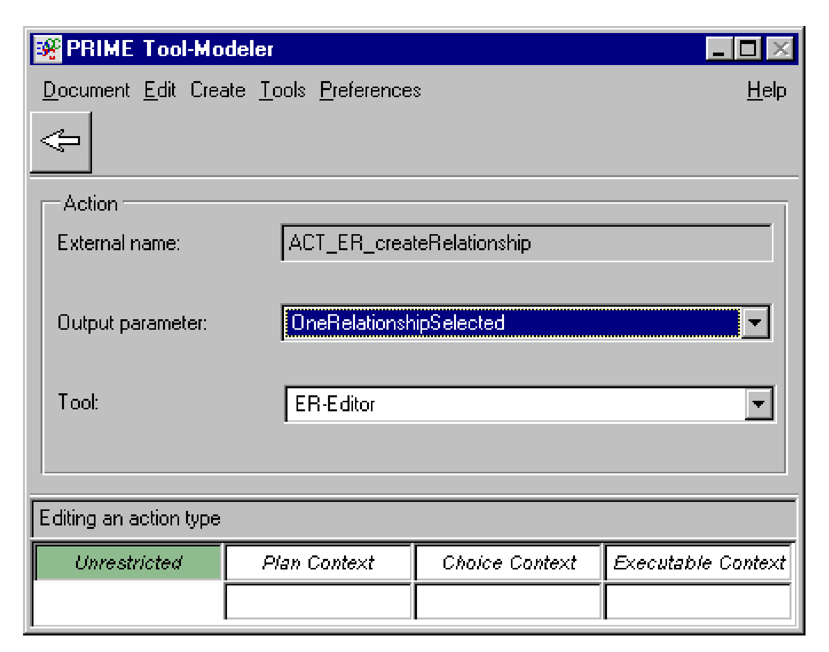
Werkzeugmodell-Editor

Mithilfe des Werkzeugmodell-Editors lassen sich Werkzeugkategorien gemäß dem in Kapitel 5 dargestellten Werkzeugmodell beschreiben. Zu der Definition einer Werkzeugkategorie gehören die Angabe der von ihr unterstützten

- Aktionen und deren Signaturen;
- Kommandoelemente (Menüs, Toolbar-Icons, Short-Keys);
- Darstellungsarten für Produkte.

Abb. 41 zeigt die beispielhaft die Modellierung einer Aktion mithilfe des Werkzeugmodell-Editors.

Abb. 41: *Der prozessintegrierte PRIME Werkzeugmodell-Editor*



Umgebungsmodell-Editor

Mithilfe des Umgebungsmodell-Editors (siehe Abb. 42) lassen sich die Querbezüge zwischen Kontextkomponenten und Werkzeugkategorien festlegen. Gemäß dem in Abschnitt 5.5 dargestellten Umgebungsmodell umfassen die Funktionalitäten des Umgebungsmodell-Editors die

- Zuordnung von Ausführungskontexten zu Werkzeugkategorien;
- Zuordnung von Entscheidungskontexten zu Werkzeugkategorien (dabei Zuordnung von Kommandoelementen zu den Alternativen eines Entscheidungskontexts);
- Überprüfung von Konsistenzbedingungen.

Abb. 42 illustriert beispielhaft, wie mithilfe des Umgebungsmodell-Editors der Entscheidungskontext CC_ER_Standard der Werkzeugkategorie ER_Editor zugeordnet wird und dabei die einzelnen Alternativen des Entscheidungskontexts an Kommandoelemente des ER_Editors gebunden werden.

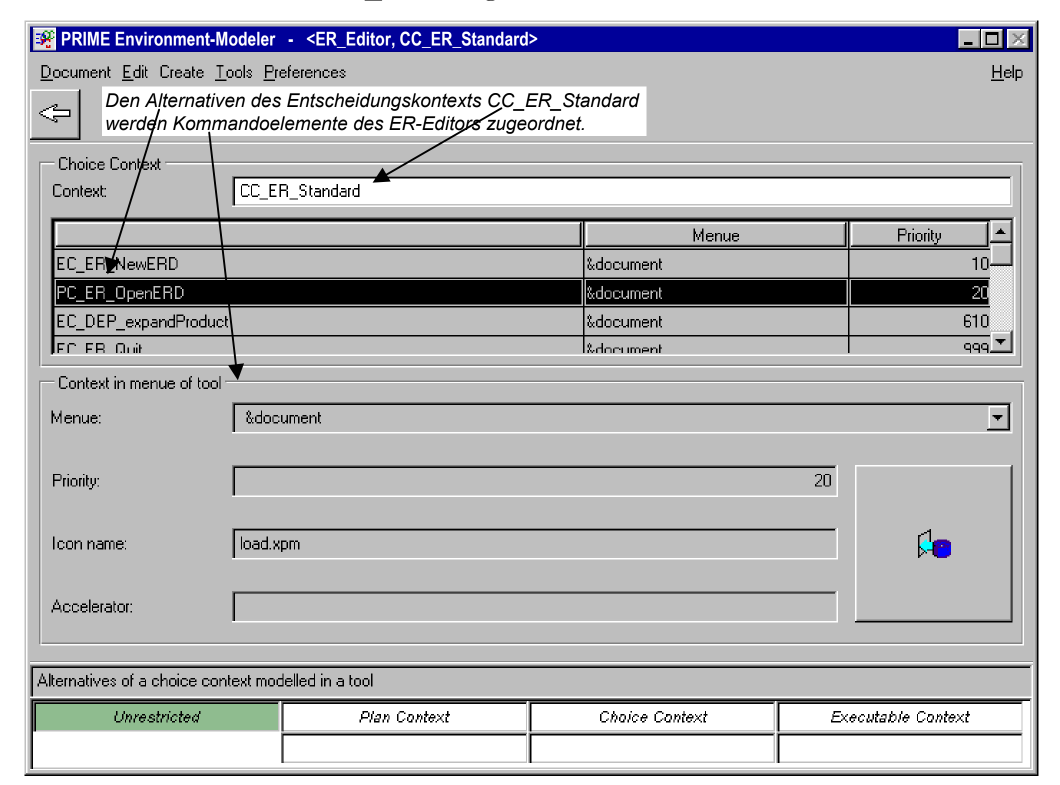


***Abb. 42**: Der prozessintegrierte PRIME-Umgebungsmodell-Editor*

7.1.2.7 Prozess-Repository

In der Modellierungsdomäne dient das Prozess-Repository der persistenten Speicherung aller Prozess-, Werkzeug- und Produktdaten. In der schematischen Darstellung des Prozess-Repositories in Abb. 34 auf Seite 170 unterscheiden wir die Teilbereiche des Repositories, die Informationen für die generischen Frameworkkomponenten vorhalten (dunkelgrau dargestellt), von solchen, die erst im Kontext spezifischer Erweiterungen sinnvoll interpretiert werden (weiß dargestellt). Daneben greifen die vorgefertigten Framework-Erweiterungen, die standardmäßig Teil jeder PRIME-basierten Umgebungen sind (Hypertext-, Entscheidungs- und Abhängigkeits-Editor sowie sprachspezifische Plankontext-Interpreter) auf die hellgrau dargestellten Repository-Bereiche zu.

Das Schema der generischen Repository-Schicht besteht aus den Teilschemata Prozessmodell, Werkzeugmodell, Umgebungsmodell und Produktmodell und realisiert die in den Kapiteln 5 und 6 definierten konzeptuellen Metamodelle[24]. Die generischen Interpreter-Komponenten der Werkzeug- und Prozessmaschinen-Frameworks sowie der Kommunikationsmanager greifen ausschließlich auf die hier abgelegten Informationen zu.

---

[24] Beachte: in diesem Kontext hat die Differenzierung zwischen generischen und spezifischen Repository-Anteilen nichts mit den unterschiedlichen Klassifikationsebenen eines Repositories im Sinne der IRDS-Architektur zu tun. Schema- und Instanzebene werden in Abb. 34 nicht explizit unterschieden!

7.2 Interaktionsprotokoll zwischen den Prozessdomänen

Das integrierte Umgebungsmodell aus Abschnitt 5.5 legt inhärent die Verantwortlichkeiten für die Ausführung von Kontexten zwischen den Prozessdomänen fest. Während Ausführungs- und Entscheidungskontexte von den Werkzeugen der Durchführungsdomäne realisiert werden, erfolgt in der Leitdomäne die Interpretation von Plankontexten. Die freie Kombinierbarkeit der drei Kontextarten durch Entscheidungskontexte (mittels der `Alternative`-Assoziation) und Plankontexte (mittels der `hat_Subkontext`-Assoziation) erfordert zur Laufzeit eine *Interaktion* zwischen den Domänen. In diesem Abschnitt definieren wir ein *Interaktionsprotokoll* zwischen der Durchführungs- und der Leitdomäne, das die wechselseitige Inanspruchnahme von Ausführungsdiensten und die Versorgung mit Rückmeldeinformation regelt. Das Interaktionsprotokoll stellt insbesondere die *Synchronisation* zwischen den Prozessdomänen sicher, d.h. es sorgt dafür, dass die Interaktionspartner jeweils korrekt auf ankommende Nachrichten reagieren, ihrerseits Nachrichten zum richtigen Zeitpunkt verschicken und somit insgesamt ihre Zustände aufeinander abgleichen.

Formalisierung des Interaktionsprotokolls durch gekoppelte UML-Zustandsdiagramme

Zur Spezifikation des Interaktionsprotokolls verwenden wir UML-Zustandsdiagramme [BoJR99]. Diese Modellierungstechnik basiert auf den von Harel eingeführten Verallgemeinerungen der Konzepte von endlichen Automaten nach Moore und Mealy [Hare87]. UML-Zustandsdiagramme fokussieren zwar eigentlich auf einzelne Objekte und dienen der Charakterisierung des *Intra*-Objektverhaltens. Für unsere Zwecke definieren wir jedoch zwei Zustandsdiagramme – je eins für die Durchführungs- und die Leitdomäne – , die über den Austausch von Nachrichten gekoppelt sind. Wir können dadurch sehr schön beschreiben, in welchen Zuständen Nachrichten verschickt werden bzw. ankommende Nachrichten sinnvoll interpretiert werden können und welche Zustandsübergänge dabei auf Seiten der Durchführungs- bzw. Leitdomäne ausgelöst werden. Das hier dargestellte Interaktionsprotokoll beruht auf den in [Pohl96; Weid95; Klam95; Poh*99] beschriebenen Ansätzen, ist jedoch umfassender und bemüht sich gleichzeitig um eine stärkere Vereinheitlichung und Vereinfachung der Zustandsdiagramme und Nachrichtentypen.

Das Zustandsdiagramm in Abb. 43 gibt eine stark vergröberte Sicht auf den Gesamtzustand einer PRIME-basierten Umgebung wieder. Wir benutzen hierbei das Strukturierungshilfsmittel der *Und-Verfeinerung*, durch das wir nebenläufige, gleichzeitig aktive Sub-Zustände darstellen können. Die Region oberhalb der gestrichelten Linie repräsentiert das Verhalten des Framework-Objekts `Tool` in der Durchführungsdomäne, während die untere Region den Zustand des Framework-Objekts `ProcessEngine` in der Leitdomäne beschreibt. Diese beiden Objekte kollaborieren miteinander, d.h. Zustandsübergänge sind in den beiden konkurrenten Teil-Zustandsdiagrammen durch den Austausch von Nachrichten (gestrichelte Pfeile) miteinander gekoppelt.

Freie und angeleitete Prozessdurchführung

Das NATURE-Prozessmodell induziert eine prinzipielle Unterscheidung zwischen zwei Zuständen einer prozessintegrierten Umgebung: *freie* und *angeleitete* Prozessdurchführung. In der freien Prozessdurchführung aktiviert der Benutzer ohne Einschränkung in seinen Werkzeugen Ausführungs- und Entscheidungskontexte. Die Prozessmaschine ist inaktiv, d.h. es findet keine Plankontext-Aus-

führung statt. In Abb. 43 wird dies durch den Tool-Zustand Unrestricted_Process_Performance und den ProcessEngine-Zustand Process_Enactment_Inactive repräsentiert. Die Benutzer-gesteuerte Aktivierung eines Plankontexts in einem Werkzeug markiert den Übergang in die angeleitete Prozessdurchführung (Guided_Process_Performance). Dieser Zustandsübergang wird der Leitdomäne durch eine GuidanceRequest()-Nachricht signalisiert, wodurch die Prozessmaschine die Interpretation des aktivierten Plankontexts startet (Process_Enactment_Active). Während der Plankontext-Ausführung kollaborieren Werkzeuge und Prozessmaschine über ContextRequest()/ContextResponse()-Nachrichtenpaare miteinander. Die Prozessmaschine kehrt nach Beendigung der Plankontext-Ausführung in den Zustand Process_Enactment_Inactive zurück und gibt dies durch eine EndOfEnactment()-Nachricht bekannt, die bei den Werkzeugen wiederum einen Zustandsübergang in den Zustand Unrestricted_Process_Performance auslöst.

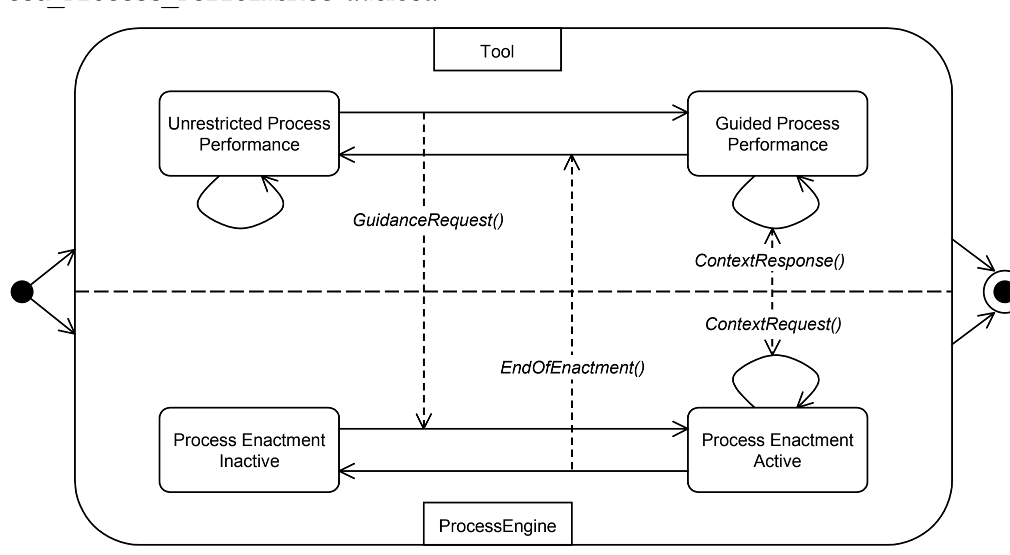


***Abb. 43**: Kopplung der Zustandsübergänge in der Durchführungs- und Leitdomäne*

Im Folgenden beschreiben wir die nebenläufigen Teil-Zustandsdiagramme der Durchführungs- und Leitdomäne im Detail. Aus präsentationstechnischen Gründen betrachten wir die beiden Domänen getrennt voneinander. Dabei sollte man jedoch im Hinterkopf behalten, dass die Zustandsübergänge in den beiden Domänen in der Regel nicht unabhängig voneinander erfolgen, sondern durch Austausch von Nachrichten aneinander gekoppelt sind.

7.2.1 Dynamische Sicht der Durchführungsdomäne

Abb. 44 zeigt das Zustandsdiagramm für die Verhaltensspezifikation eines Tool-Objekts in der Durchführungsdomäne.

Ein Werkzeug geht nach seinem Start entweder in den Zustand Standard_Context_Active oder in den Zustand Waiting_for_Context_Request über, abhängig davon, ob das Werkzeug manuell durch den Benutzer (Transition 1) oder im Rahmen einer Plankontext-Ausführung durch die Prozessmaschine (Transition 2) gestartet wurde. Diese beide Zustände sind Teil einer Oder-Verfeinerung der Zustände Unrestricted_Process_Performance bzw. Guided_Process_Performance, die wir bereits aus Abb. 43 kennen.

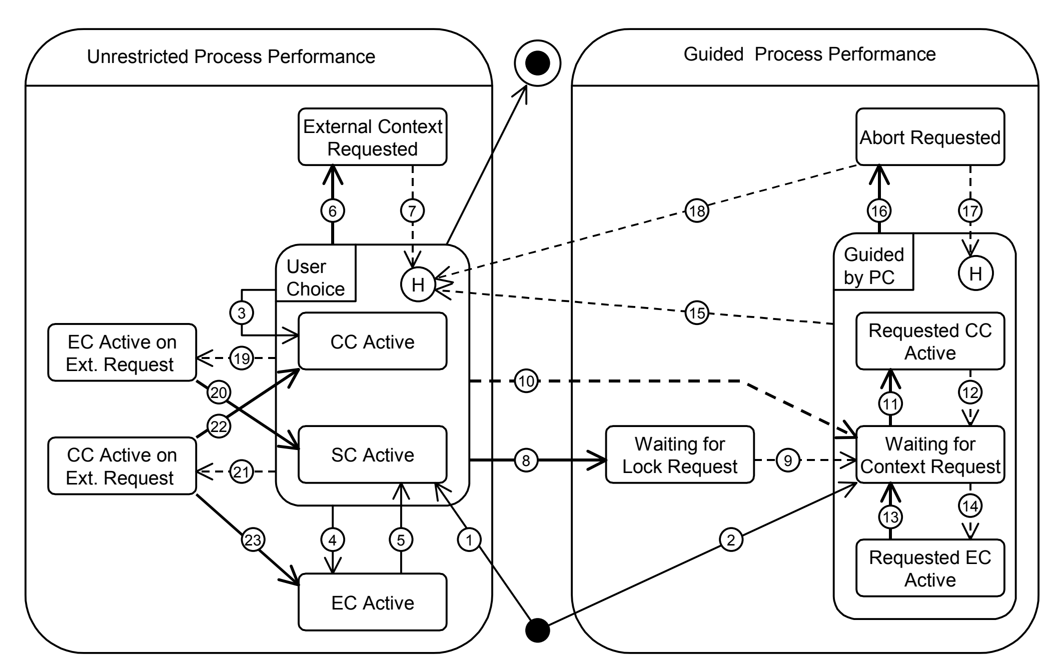


***Abb. 44**: Verhaltensspezifikation der Frameworkklasse* `Tool`

Der Super-Zustand `Unrestricted_Process_Performance` beschreibt das Werkzeugverhalten, solange die Prozessdurchführung nicht durch eine Plankontext-Interpretation angeleitet wird. Dieser Zustand besteht aus einer Oder-Verfeinerung in eine Reihe von Sub-Zuständen, deren wichtigster der Sub-Zustand `User_Choice` ist. Der Zustand `User_Choice` selbst ist seinerseits unterteilt in die Zustände `Choice_Context_Active` und `Standard_Context_Active`., die sich jedoch nur geringfügig voneinander unterscheiden. In ersterem ist ein Entscheidungskontext aktiv, den der Benutzer zuvor aktiviert hatte. Dagegen wird der Zustand `Standard_Context_Active` immer dann angenommen, wenn dem Werkzeug die aktuelle Intention des Benutzers nicht bekannt ist. Formal ist ein Standardkontext[25] allerdings nicht anderes als ein Entscheidungskontext mit einer „leeren" Intention. In beiden Sub-Zuständen nimmt das Werkzeug Benutzerinteraktionen (Selektion/Deselektion von Produkten, Aktivierung von Kommandos) entgegen und gleicht diese mit den Intentions- und Situationsdefinitionen der aktuell erlaubten Alternativen ab.

Entsprechen die aktivierten Kommandos und Produkte der Intention und Situation eines vordefinierten Kontexts `c`, wird ein `ContextMatched(c)`-Ereignis ausgelöst. Dieses Ereignis initiiert einen von mehreren möglichen Zustandsübergängen aus dem Zustand `User_Choice`. Welcher Zustandsübergang konkret ausgelöst wird, hängt von der Kontextkategorie des Kontexts `c` und seiner Zuordnung zu einer Werkzeugkategorie ab. Es sind folgende Fälle möglich:

- `c` ist ein werkzeug*interner* Entscheidungskontext.

  Dann geht das Werkzeug in den Zustand `Choice_Context_Active` über (Transition 3). Damit verbleibt das Werkzeug quasi im Super-Zustand `User_Choice`. Allerdings werden beim Wieder-Eintritt in diesen Zustand

[25] Es kann pro Werkzeugkategorie durchaus mehrere Standardkontexte geben, die sich dann durch ihren Situationsteil unterscheiden. Beispielsweise kann die Situation vorliegen, dass noch kein Produkt geladen wurde, oder die Situation, dass bereits ein Produkt eines bestimmten Typs geladen wurde.

die Interaktionsmöglichkeiten des Benutzers auf die für den Entscheidungskontext c definierten Alternativen angepasst.

- c ist ein werkzeug*interner* Ausführungskontext.

  Dann geht das Werkzeug in den Zustand `Executable_Context_Active` über (Transition 4) und führt die im Umgebungsmodell mit dem Ausführungskontext c assoziierte Werkzeugaktion aus. Nach Beendigung der Werkzeugaktion kehrt das Werkzeug in den Zustand `Standard_Context_Active` zurück (Transition 5). Dabei werden die Interaktionsmöglichkeiten des Benutzers auf die für den Standardkontext definierten Alternativen angepasst.

- c ist ein werkzeug*externer* Ausführungs- oder Entscheidungskontext,

  d.h. c ist im Umgebungsmodell einer anderen Werkzeugkategorie zugeordnet als der des Werkzeugs, in dem c aktiviert wurde. Dann geht das Werkzeug in den Zustand `External_Context_Requested` über und setzt dabei eine `ExternalECCCRequest()`-Nachricht ab (Transition 6). In diesem Zustand verbleibt das Werkzeug solange, bis es eine `ExternalECCC-Response()`-Nachricht erhält. Diese Nachricht führt zu einem Zustandsübergang (Transition 7) zurück in den Sub-Zustand von `User_Choice`, den das Werkzeug zuvor verlassen hatte (formal durch einen „History"-Pseudozustand repräsentiert). Dabei werden die Interaktionsmöglichkeiten wieder auf die für den zuvor aktiven Entscheidungs- oder Standardkontext definierten Alternativen angepasst.

- c ist ein Plankontext.

  In diesem Fall setzt das Werkzeug eine `GuidanceRequest(c)`-Nachricht an die Leitdomäne ab und geht in Erwartung der `LockRequest()`-Nachricht, die die Prozessmaschine an alle benötigten Werkzeuge aussenden wird, in den `Waiting_for_Lock_Request`-Zustand über (Transition 8). Dieser Zustandsübergang markiert den Wechsel von der freien in die angeleitete Prozessdurchführung.

Die bislang hier beschriebenen Zustandsübergänge aus dem Zustand `User_Choice` werden durch das *endogene* Ereignis `ContextMatched()` ausgelöst. Daneben definiert das Zustandsdiagramm auch die Reaktion eines Werkzeugs auf eine Reihe von *exogenen* Ereignissen, die durch asynchrone Nachrichten von außen, d.h. von anderen Werkzeugen oder von der Prozessmaschine, generiert werden:

- Transitionen 19 und 21/`ExternalECCCRequest()`:

  Diese Zustandsübergänge werden ausgelöst, wenn ein fremdes Werkzeug einen werkzeugeigenen Kontext c mittels einer `ExternalECCCRequest(c)`-Nachricht anfordert. Handelt es sich beim Kontext c um einen Ausführungskontext, geht das Werkzeug in den Zustand `EC_Active_on_External_Request` über (Transition 19) und führt die mit dem Ausführungskontext c assoziierte Aktion durch. Nach der Beendigung werden die Resultate der Aktion an das aufrufende Werkzeuge mit einer `ExternalECCCResponse()`-Nachricht zurückgeschickt, und das Werkzeug geht in den Zustand `Standard_Context_Active` über (Transition 20). Falls c ein Entscheidungskontext ist, geht das Werkzeug in den Zustand `CC_Active_on_External_Request` über (Transition 21). Hier werden dann

die Interaktionsmöglichkeiten des Benutzers auf die für c definierten Alternativen angepasst. Nach Auswahl eines alternativen Kontexts a durch den Benutzer geht das Werkzeug je nach Kontextkategorie von a in den Zustand EC_Active oder CC_Active über (Transition 22 oder 23). In beiden Fällen wird die ausgewählte Alternative dem aufrufenden Werkzeug mithilfe einer ExternalECCCResponse()-Nachricht übermittelt.

- Transition 10/LockRequest():

  Dieser Zustandsübergang wird durch den Empfang einer LockRequest()-Nachricht von der Prozessmaschine ausgelöst. Dieser Fall tritt dann ein, wenn die Prozessmaschine zuvor durch eine GuidanceRequest()-Nachricht eines *anderen* Werkzeugs aktiviert worden ist. Ein Werkzeug, das eine LockRequest()-Nachricht erhält, quittiert diese Aufforderung durch eine LockOkResponse()-Nachricht an die Leitdomäne und geht in den Waiting_for_Context_Request-Zustand. Dabei deaktiviert es alle Möglichkeiten zur Benutzerinteraktion an der Benutzeroberfläche.

Im Waiting_for_Context_Request-Zustand wartet ein Werkzeug auf ContextRequest(c)-Nachrichten durch die Prozessmaschine. Je nach dem, ob es sich beim angeforderten Kontext c um einen Ausführungs- oder Entscheidungskontext handelt, geht das Werkzeug in den Zustand Requested_EC_Active oder Requested_CC_Active über. Die Resultate der Aktionsausführung bzw. der Benutzerauswahl werden durch eine ECResponse()- bzw. CCResponse()-Nachricht an die Prozessmaschine zurückgemeldet (Transition 13 bzw. 12).

Während sich ein Werkzeug in einem der Zustände Waiting_for_Context_Request, Requested_EC_Active oder Requested_CC_Active befindet (diese Zustände sind im Super-Zustand Guided_by_PC zusammengefasst), kann der Benutzer einen Abbruch des gerade aktiven Plankontexts anfordern. Dabei setzt das Werkzeug eine AbortRequest()-Nachricht an die Prozessmaschine ab und geht in den Zustand Abort_Requested über. Falls ein Abbruch des aktuellen Plankontexts nicht möglich ist, antwortet die Prozessmaschine mit einer AbortDenied()-Nachricht und das Werkzeug kehrt zurück in den zuvor verlassenen Sub-Zustand von Guided_by_PC (Transition 17; endet im History-Pseudozustand). Im Falle eines zulässigen Plankontext-Abbruchs antwortet die Prozessmaschine mit einer AbortOK()-Nachricht. In diesem Fall kehrt das Werkzeug in den Zustand zurück, den es bei der Aktivierung des Plankontexts verlassen hatte (Transition 18). Eine ordnungsgemäße Beendigung der Plankontext-Ausführung wird dem Werkzeug durch eine EndOfEnactment()-Nachricht signalisiert. Auch hier geht das Werkzeug in den letzten Sub-Zustand vor der angeleiteten Prozessdurchführung über (Transition 15).

Tab. 10 liefert Detailinformationen zu den Zuständen und Transitionen aus Abb. 44. Für die Zustände spezifizieren wir die Aktivitäten beim Eintritt bzw. während des Verweilens in dem Zustand. Bei den Transitionen geben wir an, durch welche Ereignisse sie ausgelöst werden, welche zusätzlichen Bedingungen gegebenenfalls gelten müssen und welche Aktionen beim Zustandsübergang auszuführen sind. Die dabei verwendeten Variablen- und Argument-Bezeichner haben folgende Bedeutung: c : Kontext; this : referenziert das durch das Zustandsdiagramm beschriebene Werkzeug; result: Resultat einer Kontextausführung. Ereignisse, die dem Empfang von Nachrichten entsprechen, und Aktionen, die den Versand von Nachrichten nach sich ziehen, sind fett gedruckt.

***Tab. 10**: Detailspezifikation der Werkzeug-Zustände und Transitionen aus Abb. 44*

Zustand	Aktivität
Standard Context Active	entry : adaptUItoStandardContext(); activity: acceptUserInput(); matchContext();
Choice Context Active	entry : adaptUItoChoiceContext(c); activity: acceptUserInput(); matchContext();
Executable Context Active	activity: result = c.execute();
External Context Requested	entry : disableUserInput();
EC Active on Ext. Request	activity: result = c.execute();
CC Active on External Re-quest	entry : adaptUItoChoiceContext(c); activity: result = c.execute();
Waiting for Lock Request	entry : disableUserInput();
Waiting for Context Request	entry : disableUserInput();
Requested EC Active	activity: result = c.execute();
Requested CC Active	entry : adaptUItoChoiceContext(c); activity: result = c.execute();
Abort Requested	entry : disableUserInput();

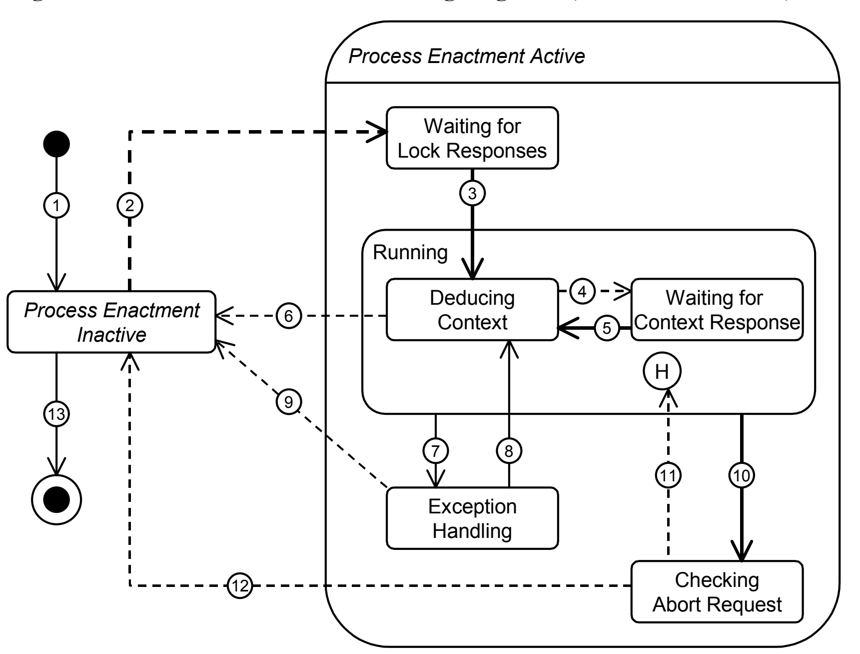
Tr.	Ereignis	Bedingung	Aktion
1	ToolStarted()	bStartedByUser = true	create();
2	ToolStarted()	bStartedByUser = false	create();
3	ContextMatched(c)	(c.ContextType == CC) and (c.Tool == this)	result = c.execute();
4	ContextMatched(c)	(c.ContextType == EC) and (c.Tool == this)	result = c.execute();
5	ContextExecuted(c, result)	(c.ContextType == EC)	
6	ContextMatched(c)	(c.ContextType != PC) and (c.Tool != this)	send **ExternalECCCRe-quest(c);**
7	**ExternalECCCResponse**(**c, result)**		
8	ContextMatched(c)	(c.ContextType == PC)	send **GuidanceRequest(c);**
9, 10	**LockRequest**()		send **LockOKResponse();**
11	**ContextRequest(c)**	(c.ContextType == CC) and (c.Tool == this)	result = c.execute();
12	ContextExecuted(c, result)	(c.ContextType == CC)	send **CCResponse(result);**
13,	**ContextRequest**(c)	(c.ContextType == EC) and (c.Tool == this)	result = c.execute();
14	ContextExecuted(c, result)	(c.ContextType == EC)	send **ECResponse(result);**
15	**EndOfEnactment**()		
16	AbortRequested()		send **AbortRequest();**
17	**AbortDenied**()		
18	**AbortOK()**		
19	**ContextRequest(c)**	(c.ContextType == EC) and (c.Tool == this)	
20	ContextExecuted(c, result)		send **ECResponse(result);**
21	**ContextRequest(c)**	(c.ContextType == CC) and (c.Tool == this)	
22	ContextExecuted(c, result)	result.ContextType = CC	send **CCResponse(result);**
23	ContextExecuted(c, result)	result.ContextType = EC	send **CCResponse(result);**
24	ToolStopped()		destroy();

7.2.2 Dynamische Sicht der Leitdomäne

Abb. 45 zeigt das Zustandsdiagramm für die Verhaltensspezifikation der Klasse ProcessEngine, die das Prozessmaschinen-Framework in der Leitdomäne repräsentiert. Nach dem Start – die Prozessmaschine wird beim Hochfahren einer PRIME-basierten Umgebung als Dämonprozess gestartet – ist die Prozessmaschine zunächst inaktiv (Zustand Process_Enactment_Inactive).

Der Empfang einer Guidance-Request()-Nachricht aus der Durchführungsdomäne löst den Übergang in den Zustand Waiting_for_Lock_Request (Transition 2) und damit in den Super-Zustand Process_Enactment_Active aus. Dieser Zustandsübergang ist mit dem Versand einer LockRequest()-Nachricht assoziiert, welche die für die Plankontext-Ausführung notwendigen Werkzeuge für die anstehende angeleitete Prozessdurchführung vorbereiten soll. Sobald die Werkzeuge durch eine LockOKResponse()-Nachricht ihre Bereitschaft signalisiert haben (Transition 3), wird für den angeforderten Plankontext ein Interpreter-Thread gestartet, und die eigentliche Plankontext-Ausführung beginnt (Zustand Running).

Abb. 45*:*
Verhaltensspezifikation der Frameworkklasse ProcessEngine



Im Zustand Deducing_Context ermittelt die Prozessmaschine den als nächstes ausführbaren Kontext c. Handelt es sich bei dem ermittelten Kontext c um einen Plankontext, wird ein weiterer Plankontext-Interpreter abgespaltet (in Abb. 45 nicht dargestellt[26]). Ansonsten (c ist ein Ausführungs- oder Entscheidungskontext) setzt die Prozessmaschine eine ContextRequest(c)-Nachricht an die Durchführungsdomäne ab (Transition 4), und geht in den Zustand Waiting_for_Context_Response über. Nach Erhalt einer ECResponse(result)- bzw. CCResponse(result)-Nachricht geht die Prozessmaschine wieder in den

[26] Im Zustand Running interagiert das übergeordnete Prozessmaschinen-Framework ständig mit dem aktuell aktiven Plankontext-Interpreter. Da das Zustandsdiagramm aus Abb. 45 auf die Interaktion des Prozessmaschinen-Frameworks mit der Durchführungsdomäne fokussiert, ist die Prozessmaschinen-*interne* Interaktion zwischen dem Framework und den einzelnen Interpreter-Threads hier nicht von Bedeutung. Wir gehen auf diesen Aspekt in Abschnitt 7.5 bei der Beschreibung des Prozessmaschinen-Frameworks noch genauer ein.

Zustand Deducing_Context über (Transition 5). Dabei wird das Ergebnis der Kontextausführung (result) an die Plankontext-Interpretation übergeben. Nach Beendigung des obersten Plankontexts innerhalb der Plankontext-Aufrufhierarchie schickt die Prozessmaschine eine EndOfEnactment()-Notifikation an die Werkzeuge der Durchführungsdomäne und kehrt in den Zustand Process_Enactment_Inactive zurück

Sollte es im Zustand Running zu einer (internen) Fehlersituation kommen, geht die Prozessmaschine in den Zustand Exception_Handling über (Transition 7). In diesem Zustand kann eine Fehlerbehandlung erfolgen. Falls der Fehler behoben werden kann, kehrt die Prozessmaschine in den Zustand Deducing_Context zurück (Transition 8), um den nächsten ausführbaren Kontext zu ermitteln. Andernfalls wird die Plankontext-Ausführung abgebrochen und die Werkzeuge durch eine EndOfEnactment()-Nachricht vom Abbruch des Plankontexts informiert (Transition 9).

***Tab. 11**: Detailspezifikation der Prozessmaschinen-Zustände und Transition aus Abb. 45*

Zustand	Aktivität
Process Enactment Inactive	activity: idle();
Waiting for Lock Response	activity: idle();
Deducing Context	activity: c = currentPCInterpreter.deduceContext();
Waiting for Context Response	activity: idle();
Exception Handling	activity: handleException();
Checking Abort Request	activity: checkAbortRequest();

Tr.	Ereignis	Bedingung	Aktion
1	ProcessEngineStarted()		create();
2	**GuidanceRequest(c)**		send **LockRequest();**
3	**LockOkResponse()**		currentPCInterpreter = new PCInterpreter(c);
4	ContextDeduced(c)	c.ContextType != PC	send **ContextRequest(c);**
5	**ECResponse(result)** or **CCResponse(result)**		
6	PCFinished()		send **EndOfEnactment();**
7	InternalError()		currentPCInterpreter. suspendEnactment();
8	ErrorChecked()	recoverable Error	currentPCInterpreter. resumeEnactment();
9	ErrorChecked()	unrecoverable Error	send **EndOfEnactment();**
10	**AbortRequest()**		currentPCInterpreter. suspendEnactment();
11	AbortChecked()	abort not possible	send **AbortDenied();**
12	AbortChecked()	abort possible	send **AbortOK();** send **EndOfEnactment();**
24	ProcessEngineStopped()		destroy();

Wird die Prozessmaschine während der Plankontext-Ausführung durch eine AbortRequest()-Nachricht aus der Durchführungsdomäne unterbrochen, geht sie in den Zustand Checking_Abort_Request über (Transition 10). In diesem Zustand kann die Prozessmaschine überprüfen, ob ein Abbruch zu Abweichungen von zwingend vorgeschriebenen Abläufen und/oder Dateninkonsistenzen führen würde. Ist der angeforderte Abbruch unzulässig, sendet die Prozessmaschine eine AbortDenied()-Nachricht an die Durchführungsdomäne und kehrt in den zuvor unterbrochenen Sub-Zustand von Running zurück (Transition 11). Andernfalls stoppt die Prozessmaschine die Plankontext-Ausführung, sendet eine AbortOK()-Nachricht an das anfragende Werkzeuge sowie eine EndOfEnactment()-Nachricht

an die übrigen Werkzeuge und geht in den Zustand `Process_Enactment_Inactive` über.

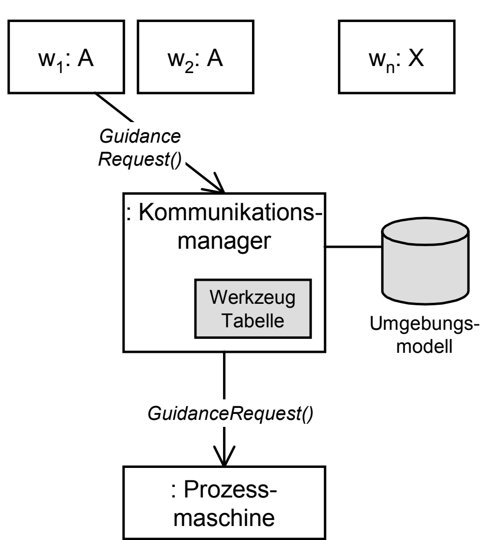
Die Zustände und Transitionen des Prozessmaschinen-Zustandsdiagramms sind in Tab. 11 mit Detailangaben über die assoziierten Aktivitäten bzw. Ereignissen, Bedingungen und Aktionen aufgelistet. Die dem Nachrichtenempfang oder -versand entsprechenden Ereignisse und Aktionen sind fett gedruckt.

7.2.3 Rolle des Kommunikationsmanagers

Kommunikationsmanager kapselt Details der Nachrichtenverteilung

Die Zustandsdiagramme, die das Verhalten von Werkzeugen und Prozessmaschine beschreiben, sind wechselseitig über den Austausch von Nachrichten gekoppelt, d.h. der Versand einer Nachricht in der Durchführungsdomäne löst ein Ereignis in der Leitdomäne aus und umgekehrt. Allerdings kommunizieren Werkzeuge und Prozessmaschine nicht direkt miteinander, sondern sind durch einen zentralen Kommunikationsmanager voneinander abgeschottet. Der Kommunikationsmanager ermöglicht den Kommunikationspartnern einen Nachrichtenaustausch auf logisch hohem Niveau, indem er das Wissen über Details der Nachrichtenverteilung kapselt. Werkzeuge und Prozessmaschine kommunizieren dabei auf der Ebene von Kontextanforderungen und -antworten, ohne die genauen Adressaten kennen zu müssen. Der Kommunikationsmanager sorgt dabei für eine Abstraktion von den *Empfängern* einer Nachricht und deren *Multiplizität*, indem er sich bei der Nachrichtenverteilung zum einen auf die Zuordnung von Kontexten zu Werkzeugkategorien im Umgebungsmodell abstützt und zum anderen in einer internen Werkzeug-Tabelle Buch über die aktuell laufenden Werkzeuginstanzen führt. Im Folgenden illustrieren wir anhand einiger charakteristischer Szenarien, wie der Kommunikationsmanager im Rahmen des oben dargestellten Interaktionsprotokolls für eine transparente Nachrichtenverteilung sorgt.

__Abb. 46__: Weiterleitung einer GuidanceRequest()-Nachricht



Aktivierung von Plankontexten

Im einfachsten Fall besteht die Funktion des Kommunikationsmanagers in einer bloßen Weiterleitung einer Nachricht. Diese Situation liegt beispielsweise dann vor, wenn ein Werkzeug nach der Aktivierung eines Plankontexts durch den Benutzer eine `GuidanceRequest()`-Nachricht absetzt. Diese Nachricht wird vom Kommunikationsmanager an die Prozessmaschine weitergeleitet (siehe Abb. 46).

Weiterleitung von Kontextanforderungen zwischen Werkzeugen

Wenn in einem Werkzeug w_1 ein Ausführungs- oder Entscheidungskontext c aktiviert wird, für den das Werkzeug nicht selbst verantwortlich ist, setzt es eine `ExternalECCCRequest()`-Nachricht ab (siehe Abb. 47). Der Kommunikationsmanager konsultiert dazu das Umgebungsmodell, um die zuständige Werkzeugkategorie zu ermitteln (hier: Werkzeugkategorie B), und schlägt in der Werkzeugtabelle nach, ob aktuell bereits eine entsprechende Werkzeuginstanz läuft (hier: w_i). Die gefundene oder eventuell neu gestartete Werkzeuginstanz erhält die `ExternalECCC-Request()`-Nachricht und führt den Kontext aus. Das Resultat der Kontextausführung wird über den Kommunikationsmanager an das aufrufende Werkzeug zurückgemeldet.

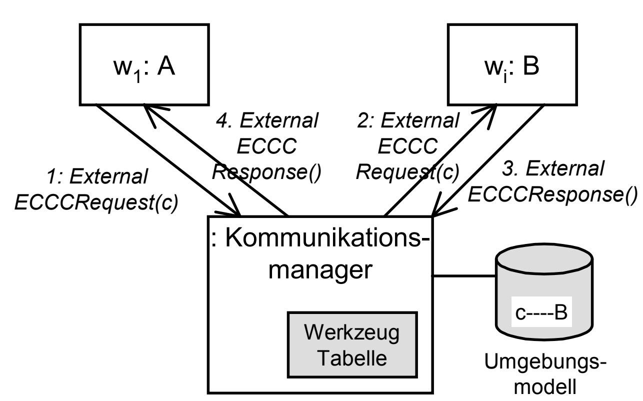


***Abb. 47**: Weiterleitung einer Kontextanforderung zwischen Werkzeugen*

Wenn man das Werkzeug-Zustandsdiagramm in Abb. 44 und die hier beschriebenen Szenarien zur Kontextaktivierung genauer betrachtet, fällt auf, dass nach dem Erkennen einer Kontextaktivierung (Ereignis `ContextMatched()`) die Werkzeuge selbst eine Vorauswahl treffen müssen, ob es sich beim dem aktivierten Kontext um einen werkzeuginternen Kontext, einen Plankontext oder einer werkzeugexternen Kontext handelt. Je nach dem, welche Situation vorliegt, wird ein interner Kontext aktiviert oder ein `GuidanceRequest()` bzw. `ExternalECCCRequest()` abgesetzt.

Vereinheitlichte Behandlung von werkzeuginternen und -externen Kontextanforderungen

Eine konzeptionell etwas elegantere Lösung läge vor, wenn die Werkzeuge überhaupt keine Kenntnis der im Umgebungsmodell definierten Zuordnung zwischen Kontexten und Werkzeugen benötigten und prinzipiell *jede* Kontextaktivierung (Ereignis `ContextMatched()`) mithilfe einer `ContextRequest()`-Nachricht zunächst an den Kommunikationsmanager meldeten. Der Kommunikationsmanager würde dann je nach Kontextzuordnung im Umgebungsmodell eine `Guidance-Request()`-Nachricht an die Prozessmaschine (Plankontext), eine `ExternalECCCRe-quest()`-Nachricht an ein anderes Werkzeug (werkzeugexterner Kontext) oder aber eine `ContextRequest()`-Nachricht zurück an das aufrufende Werkzeug (werkzeuginterner Kontext) versenden. Diese Lösung hätte also den Vorzug, dass die Zustandssteuerung der Werkzeuge wegen der vollständigen Abstraktion von der Kontextzuordnung vereinfacht werden könnte und lediglich der Kommunikationsmanager als einzige PRIME-Komponente auf die im Umgebungsmodell abgelegten Zuordnungsinformationen zugreifen müsste. Dieser Vorteil wird allerdings durch zu erwartende Performanzeinbußen relativiert, da dann auch jede

werkzeuginterne Kontextaktivierung eine Netzwerkkommunikation mit dem Kommunikationsmanager verursachen würde[27].

Sperren der benötigten Werkzeuginstanzen

Multiplexing/ Demultiplexing von Nachrichten

Zum Sperren der für eine Plankontext-Ausführung benötigten Werkzeuginstanzen muss die Prozessmaschine lediglich eine einzelne `LockRequest()`-Nachricht absetzen, die vom Kommunikationsmanager durch eine gebündelte `LockOKResponse()`-Nachricht, welche die Bereitstellung *aller* benötigten Werkzeuginstanzen signalisiert, beantwortet wird. Der Kommunikationsmanager abstrahiert aus Sicht der Prozessmaschine somit von der *Menge* der aktuell zu sperrenden Werkzeuginstanzen und sorgt für eine transparente Synchronisation der einzelnen `LockRequest()`/`LockOKResponse()`-Paare mit den jeweiligen Werkzeuginstanzen (siehe Abb. 48a). Auf ähnliche Weise erfolgt die Verteilung einer `EndOfEnactment()`-Nachricht, die die Prozessmaschine nach Beendigung der Plankontext-Ausführung absetzt (siehe Abb. 48b).

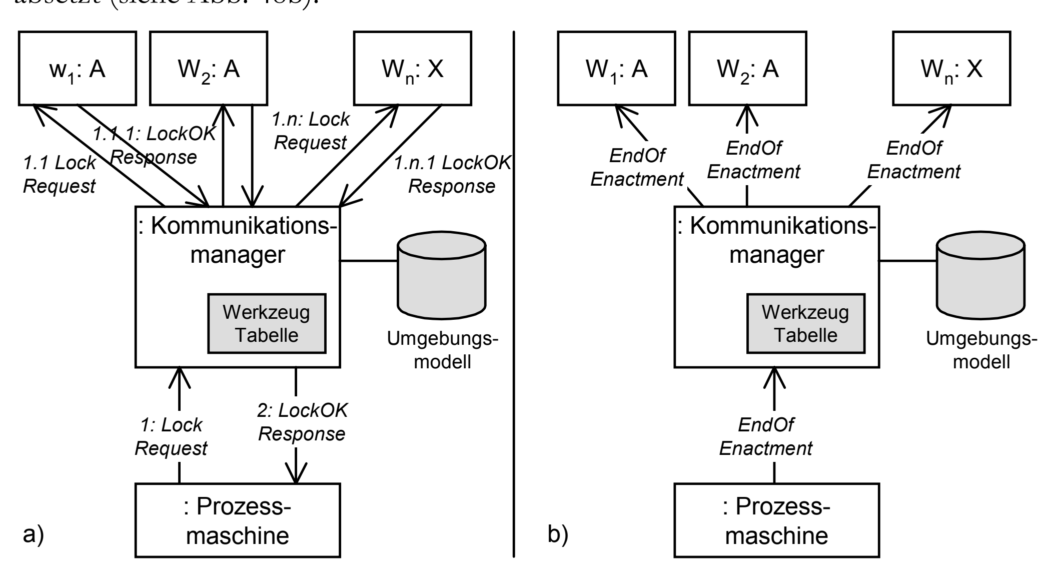


***Abb. 48**: Transparentes Sperren und Entsperren der Werkzeuginstanzen*

Kontextanforderungen durch die Prozessmaschine

Während der Plankontext-Ausführung ruft die Prozessmaschine mithilfe einer `ContextRequest()`-Nachricht einen Entscheidungs- oder Ausführungskontext `c` bei den Werkzeugen auf. Der Kommunikationsmanager ermittelt zunächst anhand der Zuordnung im Umgebungsmodell die für die Ausführung von `c` zuständige Werkzeugkategorie (hier: Werkzeugkategorie `A`; siehe Abb. 49). In der Tabelle der aktuell laufenden Werkzeuginstanzen schlägt der Kommunikationsmanager nach, ob bereits eine Instanz der entsprechenden Kategorie aktiv ist (hier: w_1 und w_2). In der Grundeinstellung leitet der Kommunikationsmanager die `ContextRequest()`-

[27] Tatsächlich haben wir in einer frühen PRIME-Version mit der hier beschriebene Alternativlösung experimentiert. Es stellte sich allerdings heraus, dass die zusätzlich erforderliche Netzwerkkommunikation mit dem Kommunikationsmanager zu mitunter störenden Verzögerungen bei der werkzeuginternen Kontextaktivierung führte. Mittlerweile verfügen wir jedoch über eine signifikant effizientere Implementierung der dem Nachrichtenprotokoll zugrunde liegenden Transportschicht, so dass Performance-Gesichtspunkte nur noch eine untergeordnete Rolle spielen dürften und eine Umstellung auf die beschriebene Alternativlösung durchaus sinnvoll erscheint.

Nachricht an die erste gefundene Instanz (hier: w_1) weiter und übermittelt die Resultate der Kontextausführung zurück an die Prozessmaschine.

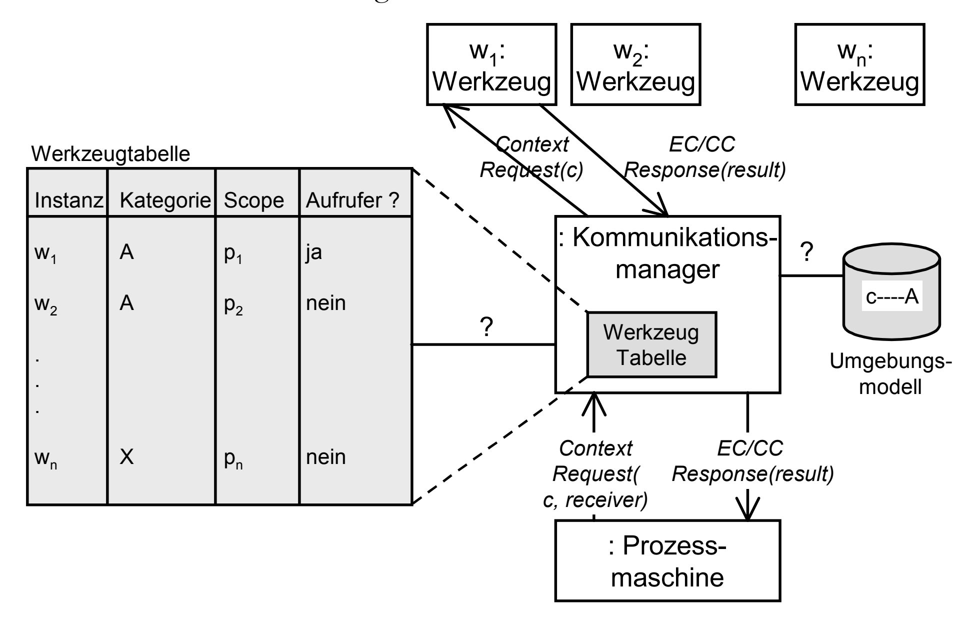


***Abb. 49**: Weiterleitung einer ContextRequest()-Nachricht*

Zu beachten ist, dass hierbei die Prozessmaschine bzw. der Modellierer des Plankontexts keinen Einfluss darauf hat, welche von möglicherweise mehreren in Frage kommenden Werkzeuginstanzen zur Laufzeit eine `ContextRequest()`-Nachricht erhält. Die Erfahrung bei der Modellierung von Plankontexten hat jedoch gezeigt, dass die an sich wünschenswerte vollständige Abstraktion von den Werkzeuginstanzen in manchen Modellierungssituationen zu Problemen führt und stattdessen eine *ablaufinhärente* Unterscheidung zwischen mehreren Instanzen der gleichen Werkzeugkategorie erforderlich ist[28].

Logische Adressierung von Werkzeuginstanzen

Die in einem Plankontext eingebetteten Subkontexte können daher mit optionaler Empfängerinformation versehen werden, die einer `ContextRequest()`-Nachricht als zusätzliches `receiver`-Attribut übergeben und vom Kommunikationsmanager bei der Nachrichtenverteilung ausgewertet wird. Dabei sind die folgenden *logischen* Adressierungsarten erlaubt:

- `callingTool`: die `ContextRequest()`-Nachricht soll an die Werkzeuginstanz übermittelt werden, die den aktuell ausgeführten Plankontext aktiviert hat. Der Kommunikationsmanager führt in der Werkzeugtabelle Buch darüber, von welcher Werkzeuginstanz der ursprüngliche `GuidanceRequest()` stammt. Im Beispiel aus Abb. 49 würde diese Adressierung zur Werkzeuginstanz w_1 aufgelöst werden. Diese Adressierungsart wird häufig dann benutzt, wenn ein komplexer, aus mehreren Schritten bestehender werkzeug*interner* Vorgang mittels eines Plankontexts beschrieben wird und durch die Prozessmaschine koordiniert werden soll.

---

[28] Ein Beispiel für eine solche Modellierungssituation ist der Plankontext `ReviseDecision`, der die Revision einer Entscheidung im Entscheidungseditor anleitet. Dabei wird in der einen Entscheidungseditor-Instanz das ursprüngliche Entscheidungsdokument angezeigt und in der anderen die Revision der Entscheidung durchgeführt. Im Zuge dieses Vorgang kann der Benutzer Positionen und Argumente der alten Entscheidung übernehmen und in das neue Entscheidungsdokument einfügen. Hierzu muss die Prozessmaschine *gezielt* mit beiden Entscheidungseditor-Instanzen interagieren.

- `newTool`: bei dieser Adressierungsart soll der Kommunikationsmanager grundsätzlich eine neue Werkzeuginstanz der entsprechende Werkzeugkategorie starten und die `ContextRequest()`-Nachricht an die gerade gestartete Instanz weiterleiten. Hierbei ist jedoch zu beachten, dass einige Werkzeugkategorien der PRIME-Umgebung (z.B. der Abhängigkeitseditor und der Produktmodelleditor) als so genannte *Singleton*-Werkzeuge deklariert wurden, d.h. es kann von diesen Werkzeugen zu einem Zeitpunkt nicht mehr als eine Instanz dieser Kategorie geben. In diesen Fällen ist die Adressierungsart `newTool` unzulässig
- `scope = ProductID`: bei dieser Adressierungsart wird die Werkzeuginstanz über einen produktbezogenen Gültigkeitsbereich (*Scope*) ermittelt. Diese Adressierungsart kann als objektorientierte Adressierung betrachtet werden. Für jede Werkzeuginstanz führt der Kommunikationsmanager Buch darüber, welche Produktinstanz die Werkzeuginstanz gerade geladen hat. Beispielsweise würde in Abb. 49 die `ContextRequest()`-Nachricht bei `scope` = p_2 an die Werkzeuginstanz w_2 weitergeleitet. Falls aktuell noch keine Werkzeuginstanz mit dem gewünschten Gültigkeitsbereich aktiv ist, startet der Kommunikationsmanager zuvor eine neue Instanz mit der entsprechenden Produktinstanz.

Mit den hier beschriebenen, *logischen* Adressierungsarten erreichen wir eine flexible Adressierung von Werkzeuginstanzen, die sich in der Praxis für die nahezu alle Modellierungssituationen als hinreichend präzise erwiesen hat. Die Vorteile der strikten Trennung zwischen logischen Kontextanforderungen und den eigentlichen Erbringern einer Kontextanforderung, die auf Architekturebene durch den Kommunikationsmanager und das Umgebungsmodell verkörpert wird, bleiben dennoch erhalten, da der Prozessmodellierer weder die Zuordnung zwischen Kontexten und Werkzeugkategorien kennen, noch eine eigene Verwaltung der aktuellen Werkzeuginstanzen vornehmen muss.

7.2.4 Zusammenfassung

Das in diesem Abschnitt beschriebene Protokoll definiert die dynamischen Beziehungen zwischen der Durchführungsdomäne und der Leitdomäne. Wir haben dazu das Verhalten der beiden Domänen mithilfe von Zustandsdiagrammen, die über Nachrichtenausausch miteinander gekoppelt sind, spezifiziert. Im Gegensatz zu der bei prozesszentrierten Umgebungen und Workflowmanagementsystemen vorherrschenden einseitigen Client-Server-Beziehung zwischen der Prozessmaschine und den Werkzeugen unterstützt das Interaktionsprotokoll sowohl einen *reaktiven* als auch einen *proaktiven* Unterstützungsmodus, wie es in den Abschnitten 2.1.5 und 3.3.5 sowie von verschiedenen Autoren (z.B. [BaDF96; CDFG96]) gefordert wird. Diese Unterstützungsmodi werden durch die Zustandspaare `Unrestricted_Process_Performance/Process_Enactment_Inactive` bzw. `Guided_Process_Performance/Process_Enactment_Active` reflektiert. Die Synchronisation der Domänen beim Übergang zwischen den Unterstützungsmodi wird durch spezielle Subprotokolle sichergestellt (z.B. durch das Sperren der benötigten Werkzeuginstanzen vor dem eigentlichen Start der Plankontext-Ausführung). Ein wichtige Rolle spielt der Kommunikationsmanager, der Details der Nachrichtenverteilung vor den

Interaktionspartnern in den beiden Domänen verbirgt und dadurch einen Nachrichtenaustausch auf einem logisch hohem Niveau ermöglicht.

7.3 GARPIT: ein Framework für prozessintegrierte Werkzeuge

Nachdem wir im vorangegangenen Abschnitt die prozessintegrierten Werkzeuge einer PRIME-basierten Umgebung aus der Perspektive ihrer externen Schnittstellen betrachtet haben, wenden wir uns nun der internen Architektur des GARPIT-Frameworks zu. Um die spezifischen Entwurfsentscheidungen besser verstehen zu können, skizzieren wir zunächst die wesentlichen funktionalen und nichtfunktionalen Anforderungen an das GARPIT-Framework (Abschnitt 7.3.1). Nach einem groben Überblick über die GARPIT-Architektur diskutieren wir einzelne Teilsysteme im Detail (Abschnitt 7.3.2).

7.3.1 Anforderungen an das GARPIT-Framework

7.3.1.1 Funktionale Anforderungen

Der integrierte Prozess- und Werkzeugmodellierungsansatz aus Kapitel 5 und das im vorangegangenen Abschnitt 7.2 definierte Interaktionsprotokoll zwischen den Prozessdomänen bilden das konzeptuelle Fundament für die Architektur eines prozessintegrierten Werkzeugs. Um eine prozessmodellkonforme Prozessdurchführung zu gewährleisten, muss das GARPIT-Framework die folgenden *funktionalen* Anforderungen in Form generischer Softwarebausteine umsetzen [PoWe97; Poh*99]:

- *Interpretation des Umgebungsmodells:*

  Das Werkzeugverhalten wird maßgeblich durch die *externen*, im Prozess-Repository abgelegten Kontextdefinitionen im Umgebungsmodell gesteuert. Um die in Abschnitt 3.1.4 geforderte *Modellierungsadaptabilität* zu erfüllen, reicht es nicht aus, diese Spezifikation manuell oder durch Codegenerierungstechniken in entsprechende Werkzeugfunktionalität zu überführen. Stattdessen ist eine Komponente erforderlich, die die externen Kontextdefinitionen *zur Laufzeit interpretiert*. Die Interpreterkomponente muss dazu folgende Teilaufgaben erfüllen:

  - Steuerung von Ausführungskontexten gemäß Umgebungsmodell:

    Nach der Aktivierung eines Ausführungskontexts muss die Interpreter-Komponenten die im Umgebungsmodell assoziierte Werkzeug-Aktion ermitteln.

  - Steuerung von Entscheidungskontexten gemäß Umgebungsmodell:

    Nach der Aktivierung eines Entscheidungskontexts muss die Benutzeroberfläche des Werkzeugs entsprechend angepasst werden, d.h. die Selektierbarkeit und Darstellung von Produkten und Kommandoelementen richtet sich nach den im Umgebungsmodell definierten Alternativen des Entscheidungskontexts.

- Erkennung der Kontextaktivierung durch den Benutzer:

  Im Gegensatz zu traditionellen Werkzeugarchitekturen ist die Reaktion eines GARPIT-basierten Werkzeugs auf Benutzereingaben nicht unmittelbar an die Implementierung von Aktionen gebunden (etwa durch Callback-Mechanismen). Stattdessen muss die Interpreter-Komponente die Interaktionen des Benutzers (Selektion von Produkten und Kommandos) mit den Kontextdefinitionen des Umgebungsmodells abgleichen und die Aktivierung eines Kontextes erkennen.

- Verwaltung von Kontextdefinitionen:

  Da potenziell jede Benutzerinteraktion mit den externen Kontextdefinitionen abgeglichen werden muss, bergen ständige Anfragen an das Prozessrepository die Gefahr eines Leistungsengpasses. Innerhalb der Werkzeugarchitektur wird daher ein Gedächtnis-Baustein benötigt, der die für ein Werkzeug relevanten Kontextinformationen aus dem Prozess-Repository in einer Laufzeit-Datenstruktur organisiert und effizient verwaltet.

- Synchronisation mit der Leitdomäne:

  Das Werkzeug muss über eine Schnittstelle zur Leitdomäne verfügen und Nachrichten mit der Prozessmaschine gemäß dem Interaktionsprotokoll aus Abschnitt 7.2 austauschen und verarbeiten können.

Neben Komponenten für die Umsetzung dieser generischen Anforderungen muss das GARPIT-Framework darüber hinaus Variationspunkte für die Erweiterung um werkzeugspezifische Produktmodelle und Benutzeroberflächenelemente vorsehen.

7.3.1.2 Nichtfunktionale Anforderungen

Zusätzlich zu den oben genannten funktionalen Anforderungen haben eine Reihe *nichtfunktionaler* Anforderungen die Architektur des GARPIT-Frameworks maßgeblich beeinflusst:

- *Wiederverwendung*:

  Wiederverwendung sowohl auf Kodeebene als auch auf Architekturebene ist die eigentliche Grundmotivation für die Wahl einer Framework-basierten Entwurfsmethodik. Wie in Abschnitt 7.1.1 dargestellt, besteht das Ziel darin, möglichst viele Gemeinsamkeiten zwischen unterschiedlichen Werkzeugen zu identifizieren, diese als allgemein verwendbares Werkzeuggerüsts heraus zu faktorisieren und wohldefinierte Variationspunkte für das „Einklinken“ spezifischer Bausteine vorzugeben.

- *Wartbarkeit und Erweiterbarkeit*:

  Wartbarkeit und Erweiterbarkeit schlägt sich an unterschiedlichen Stellen der Architektur u.a. durch Schichtenbildung und durch den Einsatz von Entwurfsmustern nieder. Beispielsweise haben wir die Zustände und Transitionen des Werkzeug-Zustandsdiagramms aus Abschnitt 7.2 mit Hilfe einer Erweiterung des *State*-Entwurfsmusters [GHJV95] umgesetzt, wodurch Umkonfigurierungen und Erweiterungen des Interaktionsprotokolls sogar zur Laufzeit sehr einfach zu realisieren sind. Diese und ähnliche Entwurfs-

entscheidungen werden wir weiter unten bei der Diskussion der einzelnen Teilsysteme noch eingehend behandeln. Des weiteren haben wir im Sinne einer „Multi-Tier"-Architektur [Ecke95] auf eine strikte Trennung zwischen Präsentationskomponenten, Produktmodellkomponenten und Ablauflogik geachtet, wobei letztere ja ohnehin außerhalb der Werkzeuge in einfach anpassbaren Prozessmodellen hinterlegt wird. Eine etwas andere Art von Schichtenbildung bestimmt das Zusammenspiel zwischen Framework-eigenen Komponenten und Softwarekomponenten, die wir von Fremdherstellern beziehen. Mithilfe von Adapterklassen abstrahieren wir von den Schnittstellen spezifischer Datenbankmodelle und -produkte, GUI-Bibliotheken und Kommunikationsmechanismen, wodurch wir gleichzeitig die Portabilität des Frameworks auf eine andere Systemplattformen sicher stellen.

- *Stabilität*:

  An Framework-Komponenten sind besondere Anforderungen hinsichtlich weitgehender Fehlerfreiheit und Stabilität zu stellen, da diese das Rückgrat einer ganzen Familie von Anwendungen darstellen und kritische Funktionsbausteine realisieren. In der Basisschicht des GARPIT- und des GARPEM-Frameworks haben wir daher Mechanismen des Vertragsmodells nach Meyer (*Design by Contract* [Meye97]) verankert, die durchgängig im gesamten Framework verwendet werden. Im Laufe der Entwicklung and Anwendung des Framework konnten wir so sehr schnell Verletzungen von Vor- und Nachbedingungen im Framework sowie in den erstellten Werkzeugen aufspüren und beheben. Darüber hinaus tragen die spezifizierten Vor- und Nachbedingungen zu einer zusätzlichen Dokumentation der Framework-Klassen und deren Methoden bei.

- *Performanz*:

  Die Forderung nach Performanz steht häufig im Konflikt zu den bisher genannten Zielen, da interpretative Mechanismen und die Bildung von Abstraktionsschichten und Indirektionen, die Komponenten voneinander entkoppeln sollen, leicht zu inakzeptablen Antwortzeiten führen, so dass bisweilen Kompromisse auf Kosten von Funktionalität und Wartbarkeit eingegangen werden müssen. In diesem Zusammenhang werden wir weiter unten auf einige Entwurfsentscheidungen genauer eingehen, hinter denen zum Teil Performanzüberlegungen stecken.

7.3.2 Architektur

7.3.2.1 Darstellung

Abb. 50 gibt einen Überblick über die wesentlichen Teilsysteme des GARPIT-Frameworks. Zur Darstellung der Architektur auf dieser relativ groben Detaillierungsebene verwenden wir zunächst UML-Paketdiagramme. Später werden wir einzelne Teilsysteme genauer diskutieren und mithilfe von Klassendiagrammen verfeinern. Die grau dargestellten Pakete bilden den generischen Anteil des GARPIT-Frameworks. In diesen Paketen sind sowohl direkt nutzbare Klassen enthalten, die allgemein verwendbare Funktionalität fertig implementieren, als auch abstrakte Klassen, die hauptsächlich Schnittstellen für den Anschluss spezifischer

Funktionalität definieren und in spezifischen Werkzeugen durch konkrete Klassen spezialisiert werden müssen. Die werkzeugspezifischen Klassen sind in den weiß dargestellten Paketen zusammengefasst. Zwischen den Paketen lassen sich drei Arten von Beziehungen identifizieren:

- *allgemeine Importbeziehungen*: diese Beziehung drückt aus, dass sich das importierende Paket auf Dienste des importierten Pakets abstützt;
- *Spezialisierungsbeziehungen*: diese Beziehung definiert die Variationspunkte des Frameworks, indem sie den Bezug zwischen Paketen, die abstrakte Framework-Klassen beinhalten, und werkzeugspezifischen Paketen herstellt. Spezialisierungsbeziehungen können als eine besondere Art von Importbeziehungen aufgefasst werden, da die Klassen des spezielleren Pakets via Vererbung Dienste von den Klassen des allgemeineren Pakets in Anspruch nehmen.
- *Rückruf-Beziehungen*: Einige Pakete generieren asynchron Ereignisse, von denen Pakete, die in der Importhierarchie der Architektur höher angesiedelt sind, notifiziert werden müssen. Eine solche Notifikation erfolgt üblicherweise über den Aufruf von Rückruf-Operationen oder -Objekten (*Callbacks*), die die an den Ereignissen interessierten Komponenten zuvor bei der Ereignis-produzierenden Komponente registriert haben. Die dadurch induzierte, erst zur Laufzeit entstehende Beziehung zwischen dem Ereignis-produzierenden und Ereignis-empfangenden Paket deklarieren wir in der Architektur als Rückruf-Beziehung. Wir vermeiden dadurch unerwünschte zyklische Importabhängigkeiten; insbesondere werden die Signaturen der Ruckrufoperationen bzw. der Typ der Rückrufobjekte im Ereignis-produzierenden Paket definiert, so dass keine wechselseitigen Compilezeit-Abhängigkeiten entstehen.

7.3.2.2 Teilsysteme in Überblick

Werkzeugspezifischer Kern

Der werkzeugspezifische Kern eines GARPIT-basierten Werkzeugs wird in den Teilsystemen `T_Model`, `T_GUI` und `T_Actions` realisiert[29]. Das Paket `T_Model` liefert eine objektorientierte Zugriffsschicht auf das logische Produktmodell, auf dem ein Werkzeug operiert. Die Persistierung des Produktmodells im Prozessrepository erfolgt mithilfe werkzeugspezifischer Anfrage- und Updateklassen aus dem Paket `T_DBInterface`. Diese Klassen sind Spezialisierungen allgemeiner Datenbankklassen im Paket `GenericDBInterface`, die vom Datenmodell und den Besonderheiten der Klient-APIs des zugrunde liegenden Datenbankmanagementsystems abstrahieren. Das Teilsystem `T_GUI` stellt die für die Präsentation erforderlichen Benutzeroberflächenkomponenten bereit.

Trennung zwischen Präsentationsebene und logischer Produktebene

Eine wichtige Entwurfsentscheidung des GARPIT-Frameworks besteht somit in der strikten Trennung zwischen den beiden Paketen `T_Model` und `T_GUI`. Dadurch bleibt das logische Produktmodell stabil gegenüber Änderungen in der externen Präsentation oder dem Austausch des verwendeten GUI-Toolkits.

---

[29] Das Präfix `T` steht hier für ein beliebiges Werkzeug. In einem konkreten Werkzeug wird dieses Kürzel durch den entsprechenden Werkzeugnamen ersetzt.

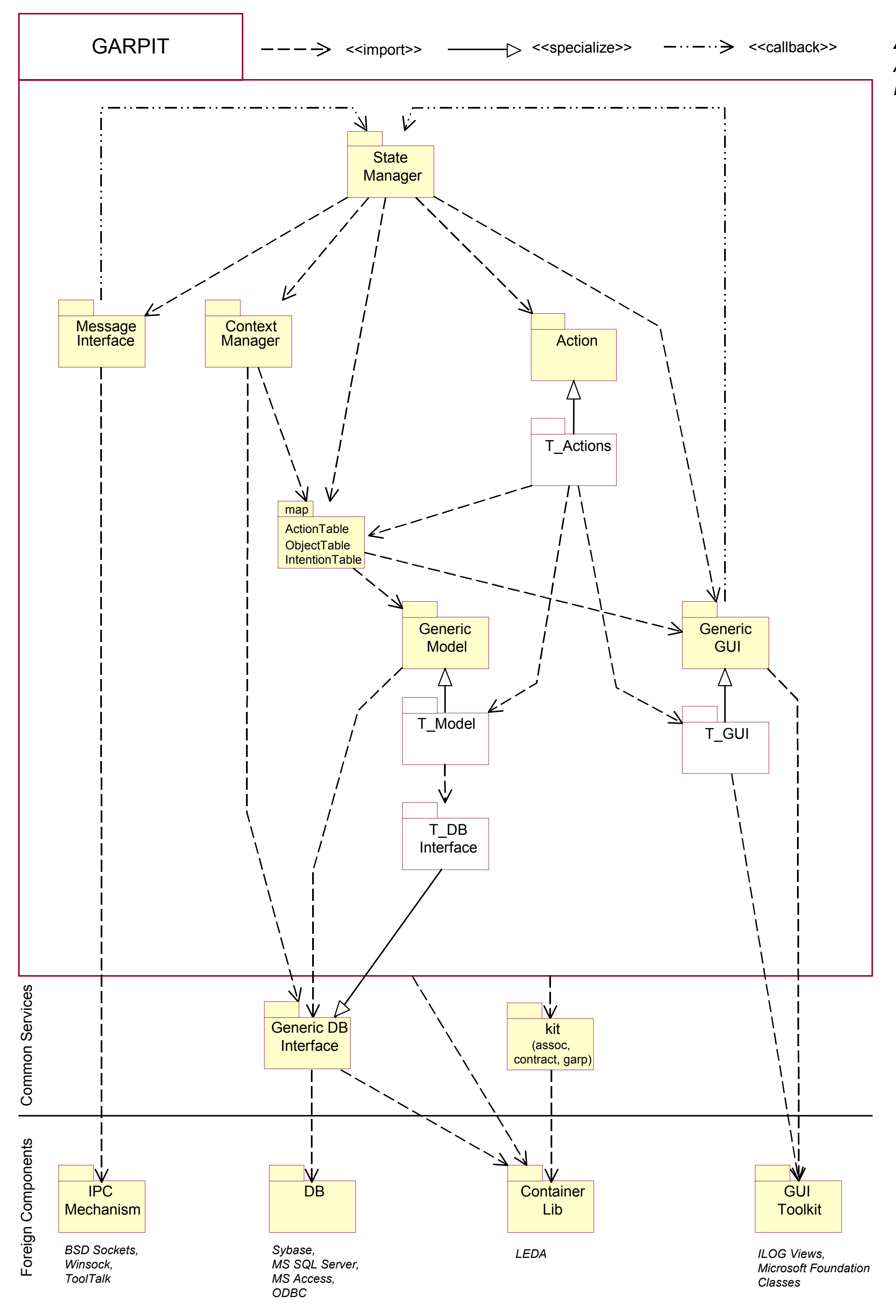


***Abb. 50**: Architektur des GARPIT-Frameworks*

T_Actions: Realisierung der Werkzeugaktionen

Das Paket `T_Actions` implementiert die im Werkzeugmodell definierten Aktionen eines Werkzeugs. Aktionen werden als eigenständige Klassen mit einer `execute()`-Methode realisiert, die Erzeugungs-, Änderungs- und Löschoperationen auf dem internen Produktmodell und der Benutzeroberfläche ausführen und somit die Präsentationsebene (`T_GUI`) mit der internen, logischen Produktmodellebene (`T_Model`) verknüpfen. Die Aktionen verfügen über kein eigenes Gedächtnis und fungieren lediglich als so genannte *Kontrollobjekte* [JCJÖ92] oder *funktionale Module*

[Nagl90]. Die durch die Aktionen realisierte Art der Verknüpfung zwischen Produktmodell und Benutzeroberfläche entspricht dem *Mediator*-Entwurfsmuster [GHJV95]. Im Vergleich zum für diesen Zweck ebenfalls häufig eingesetzten *Observer*-Entwurfsmuster hat dieser Ansatz den Vorteil, dass keinerlei Abhängigkeiten zwischen Benutzeroberfläche und Produktmodell auftreten.

Um zu einem `T_Model`-Objekt das entsprechende `T_GUI`-Objekt wiederzufinden (und umgekehrt), stützen sich die Aktionen auf die Klasse `ObjectTable` des Pakets `Map` ab, die ein Verzeichnis für die Zuordnung zwischen `T_GUI`- und `T_Model`-Objekten bereitstellt (mehr dazu weiter unten).

Anbindung an das Framework über abstrakte Oberklassen

Die Anbindung der werkzeugspezifischen Pakete `T_Model`, `T_GUI` und `T_Actions` an das generische Framework erfolgt über die korrespondierenden Pakete `GenericModel`, `GenericGUI` und `GenericActions`. Diese Pakete definieren die *Variationspunkte* des Frameworks in Form abstrakter oder semiabstrakter Klassen, deren Schnittstellen (virtuelle Methoden) den generischen Framework-Paketen bekannt sind und die in den werkzeugspezifischen Paketen verfeinert bzw. implementiert werden müssen.

ContextManager: Interpretation des Umgebungsmodells

Der `ContextManager` realisiert einen Interpreter für den werkzeugrelevanten Anteil des Umgebungsmodells, d.h. er gleicht Benutzerinteraktionen (Produkt- und Kommandoselektionen) mit den Kontextdefinitionen ab und steuert die Ausführung von Ausführungs- und Entscheidungskontexten (siehe Anforderung „Interpretation des Umgebungsmodells" aus Abschnitt 7.3.1). Der `ContextManager` baut seine internen Laufzeitdatenstrukturen komplett aus den im Prozess-Repository abgelegten Modellen auf. Das bedeutet insbesondere, dass der `ContextManager` Produkte, Intentionen und Aktionen ausschließlich über die entsprechenden eindeutigen Identifikatoren im Prozess-Repository referenziert und nicht direkt auf den `Model`-, `GUI`- und `Action`-Klassen eines spezifischen Werkzeugs operieren darf. Dies erfordert zwar eine zusätzliche Indirektion zwischen dem `ContextManager` und den werkzeugspezifischen Klassen, ermöglicht uns jedoch, den `ContextManager` als vollständig generisches Teilsystem zu realisieren.

Map: Verknüpfung zwischen Laufzeitobjekten und Repository-Identifikatoren

Die Indirektion wird durch die drei Verzeichnisklassen `ObjectTable`, `IntentionTable` und `ActionTable` realisiert, die im Paket `Map` angesiedelt sind. Diese Klassen verwalten die Korrespondenzen zwischen den logischen Repository-Identifikatoren und den entsprechenden Laufzeit-Objekten eines spezifischen Werkzeugs. In der `ActionTable` registrieren sich die `T_Action`-Objekte eines Werkzeugs und können vom `ContextManager` über die korrespondierenden Identifikatoren des Umgebungsmodells aufgefunden und aktiviert werden. Dadurch benötigt der `ContextManager` keinen hartkodierten Bezug zu den werkzeugspezifischen `T_Action`-Klassen. In der `ObjectTable` werden Informationen über die aktuell geladenen Produkte in Instanzen der Klasse `ObjectDescriptor` verwaltet, wobei jeder `ObjectDescriptor` den eindeutigen Repository-Identifikator eines Produkts mit Referenzen auf die entsprechenden `T_Model`- bzw. `T_GUI`-Laufzeitobjekte verknüpft. Auf ähnliche Weise assoziiert die `IntentionTable` logische Intentions-Identifikatoren mit entsprechenden Kommandoelementen aus dem Paket `T_GUI`. Änderungen des Selektions- bzw. Selektierbarkeitsstatus von Produkten und Kommandoelementen werden zwischen dem `ContextManager` und der Präsentationsschicht ausschließlich über `ObjectTable` und `IntentionTable` ausgetauscht. Die `ObjectTable` und die `IntentionTable` liefern dem `ContextManager` somit eine abstrakte Sicht auf den aktuellen Werkzeugzustand, die völlig unabhängig ist von

den Klassen des werkzeuginternen Produktmodells und der Benutzeroberfläche ist.

StateManager: globale Zustandskontrolle gemäß Interaktionsprotokoll

Das generische Teilsystem StateManager realisiert die globale Werkzeugkontrolle gemäß dem in Abschnitt 7.2 definierten Zustandsdiagramm. Ereignisse, die im StateManager verarbeitet werden, werden von den Teilsystemen MessageInterface und GenericGUI generiert.

MessageInterface: Kommunikationskanal zwischen Durchführungs- und Leitdomäne

Das Teilsystem MessageInterface etabliert einen Kommunikationskanal zur Leitdomäne (eigentlich zum Kommunikationsmanager) und erlaubt das Verschicken sowie den synchronen und asynchronen Empfang der in Abschnitt 7.2 definierten Nachrichten. Es abstrahiert damit vom konkret eingesetzten Mechanismus zur Interprozess-Kommunikation (IPC) und erlaubt dessen einfache Austauschbarkeit.

GenericGUI: Oberflächenprogrammierung

Das Teilsystem GenericGUI stellt eine Reihe allgemein verwendbarer Klassen für Fenster, Menüverwaltung, Dialoge und grafische Objekte (Shapes) zur Verfügung und abstrahiert dabei vom zugrunde liegenden GUI_Toolkit. Es sorgt insbesondere für die Weiterleitung von Benutzerereignissen an den StateManager, so dass der StateManager (und die restlichen generischen Architekturkomponenten) unabhängig vom gewählten GUI_Toolkit realisiert werden kann. Werkzeugspezifische Benutzeroberflächen (T_GUI) kommen weitgehend mit den von GenericGUI bereitgestellten Diensten aus und müssen nur in Ausnahmefällen auf Klassen eines spezifischen GUI_Toolkits zugreifen[30].

Kit: Programmieren-per-Vertrag, Assoziationsklassen, Attributrepräsentation

Das Teilsystem Kit ist eine Sammlung von allgemein verwendbaren Hilfsklassen, die von vielen Teilsystem des GARPIT-Frameworks verwendet werden. Neben einer Reihe von Typabstraktionen enthält dieses Paket vor allem die Teilpakete Contract, Assoc und GARP. Contract enthält Hilfsmittel für das *Programmieren-per-Vertrag* (Design-by-Contract [Meye97]). Assoc stellt eine Familie von Template-Klassen für die explizite Verwaltung von Assoziationen zwischen Klassen bereit, ohne diese durch unidirektionale Zeiger realisieren zu müssen. GARP[31] ist eine Klassenbibliothek für die Laufzeitrepräsentation hierarchisch strukturierter Attributbäume und deren Serialisierung/Deserialisierung in eine Zeichenketten-Darstellung. GARP ähnelt von seiner Funktionalität sehr stark dem Document Object Model (DOM) für die Laufzeitdarstellung von XML-Daten [W3C#98] und wird hauptsächlich für das Marshalling/Unmarshalling von Nachrichtenparametern im Teilsystem MessageInterface verwendet.

Im Folgenden stellen wir die wichtigsten Pakete des GARPIT-Frameworks im Detail vor. Wir beschränken uns dabei auf die generischen Teilsysteme StateManager, MessageInterface und ContextManager, da gerade diese Teilsysteme das wesentliche Unterscheidungsmerkmal unseres Ansatzes von anderen Vorschlägen für generische Werkzeugarchitekturen (z.B. [Emme95; Lefe95]) darstellen.

---

[30] Die komplette Verkapselung eines nativen GUI_Toolkits durch entsprechende Adapterklassen ist in der Regel mit einem sehr hohen Aufwand verbunden, der auch durch die dadurch gewonnene Portabilität meist nicht gerechtfertigt wird [Ble*99; Ble*99a]. Wir haben uns daher bewusst auf die Kaspelung der am häufigsten benötigten Oberflächenelemente (Standarddialoge, Shapes, Container, Menüsystem) im Paket GenericGUI beschränkt.

[31] GARP: **G**eneral **A**ttribute **R**e**P**resentation

7.3.2.3 StateManager

Das globale Werkzeugverhalten in Reaktion auf Nachrichtenereignisse aus der Leitdomäne und Benutzerinteraktionen wurde in Abschnitt 7.2 mithilfe eines Zustandsdiagramms spezifiziert und muss nun innerhalb der Werkzeugarchitektur umgesetzt werden. In der Praxis erfolgt die Abbildung solcherart spezifizierter Dynamikaspekte in einen Klassenentwurf meist nicht auf besonders systematische Weise. Generell werden Zustandsdiagramme eher benutzt, um auf konzeptueller Ebene das gewünschte Verhalten zu klären und dieses dann in korrespondierende, maßgeschneiderte Codefragmente zu übertragen, die häufig über Methoden vieler verschiedener Klassen in unterschiedlichen Systemschichten verstreut sind. Dabei geht jedoch die Sichtbarkeit der Zustände und Transitionen im Klassendesign und in der Implementierung verloren. Dies hat zur Folge, dass bei Protokolländerungen die betroffenen Stellen einer Architektur nur schwer zu identifizieren sind [HiKa99].

Übertragung des Interaktionsprotokolls in explizite Zustands- und Transitionsklassen

Um Wartbarkeit und Erweiterbarkeit des globalen Werkzeugverhaltens in Hinblick auf zukünftige Anforderungen zu gewährleisten, werden im GARPIT-Framework die in Abschnitt 7.2 definierten Zustände und Transitionen als *explizite Klassen* in die Architektur übertragen. Abb. 51 zeigt die resultierende Detailstruktur des `StateManager`-Teilsystems auf Klassenebene. Jeder Zustand und jede Transition wird als Subklasse der abstrakten Klassen `CsttState` bzw `CsttTransition`[32] realisiert. Die Topologie eines aus spezifischen `CsttState`- und `CsttTransition`-Objekten aufgebauten Zustandsdiagramms wird innerhalb der Klasse `CsttStateDriver` organisiert.

Kontrollfunktionalität in den Methoden der Zustands- und Transitionsklassen

Die Zustands- und Transitionsklassen steuern den Aufruf der übrigen GARPIT-Komponenten, z.B. in den Teilsystemen `ContextManager` oder `MessageInterface`. Diese Kontrollfunktionalität wird entsprechend dem UML-Standard für Zustandsdiagramme in den Methoden `doEntry`, `doActivity` und `doExit` (beim Eintritt in, Verweilen in und Verlassen von Zuständen) bzw. `doAction` (während einer Transition) hinterlegt. Diese Methoden sind als virtuelle, leere Methoden in den Klassen `CsttState` bzw. `CsttTransition` vordefiniert und werden bei Bedarf in den jeweiligen Subklassen redefiniert und mit spezifischer Funktionalität gefüllt. Beispielsweise ruft die `doEntry`-Methode der `CsttWaitingForLockRequest`-Klasse die `disableUserInput`–Methode des Teilsystems `GenericGUI` auf. Jede `CsttTransition`-Subklasse verfügt weiterhin über geerbte und evt. überschriebene `checkEvent`- und `checkCondition`-Methoden. Mit diesen Methoden realisiert eine `CsttTransition`-Klasse den Test, ob das aktuell anliegende Ereignis die Transition schalten kann und ob die eventuell zusätzlich geforderten Bedingungen gelten. Tab. 10 aus Abschnitt 7.2.1 gibt einen Überblick über die Funktionalität der einzelnen Werkzeugzustände und -Transitionen.

Die Klasse `CsttStateDriver` realisiert eine generische Zustandsmaschine und wurde als *Singleton*-Klasse entworfen, d.h. pro Werkzeuginstanz existiert genau eine Instanz dieser Klasse, die beim Werkzeugstart kreiert wird. Das `CsttStateDriver`-Objekt verwaltet die Verknüpfungen zwischen einer Menge von `CsttState`- und

---

[32] Alle Klassennamen des PRIME-Frameworks tragen ein Präfix `C<Paketkürzel>`, das sich auf das Paket, dem die Klasse zugeordnet ist, bezieht. Das `StateManager`-Paket ist intern in zwei Teilpakete aufgeteilt: `State` mit dem Präfix `Cstt` und `Event` mit dem Präfix `Cevt`.

CsstTransition-Objekten, d.h. die Beziehungen zwischen CsttState- und CsttTransition-Objekten sind a priori nicht hartkodiert vorgegeben. Als Basisklasse für spezialisierte StateDriver-Klassen implementiert der CsttStateDriver zunächst nur ein leeres Zustandsdiagramm. Für ein spezifisches Zustandsdiagramm wird eine Subklasse von CsttStateDriver definiert, die in ihrem Konstruktor die benötigten CsttState- und CsttTransition-Objekte selbst erzeugt und eine entsprechende Verknüpfungsstruktur initialisiert. Innerhalb des GARPIT-Frameworks übernimmt die Klasse CsttToolStateDriver diese Rolle. Auf Seiten des Prozessmaschinen-Frameworks GARPEM gibt es eine korrespondierende Klassenfamilie (CsttProcessEngineStateDriver mit CsttProcessEngineState- und CsttProcessEngineTransition-Subklassen), die ebenfalls von den hier skizzierten Basisklassen abgeleitet ist.

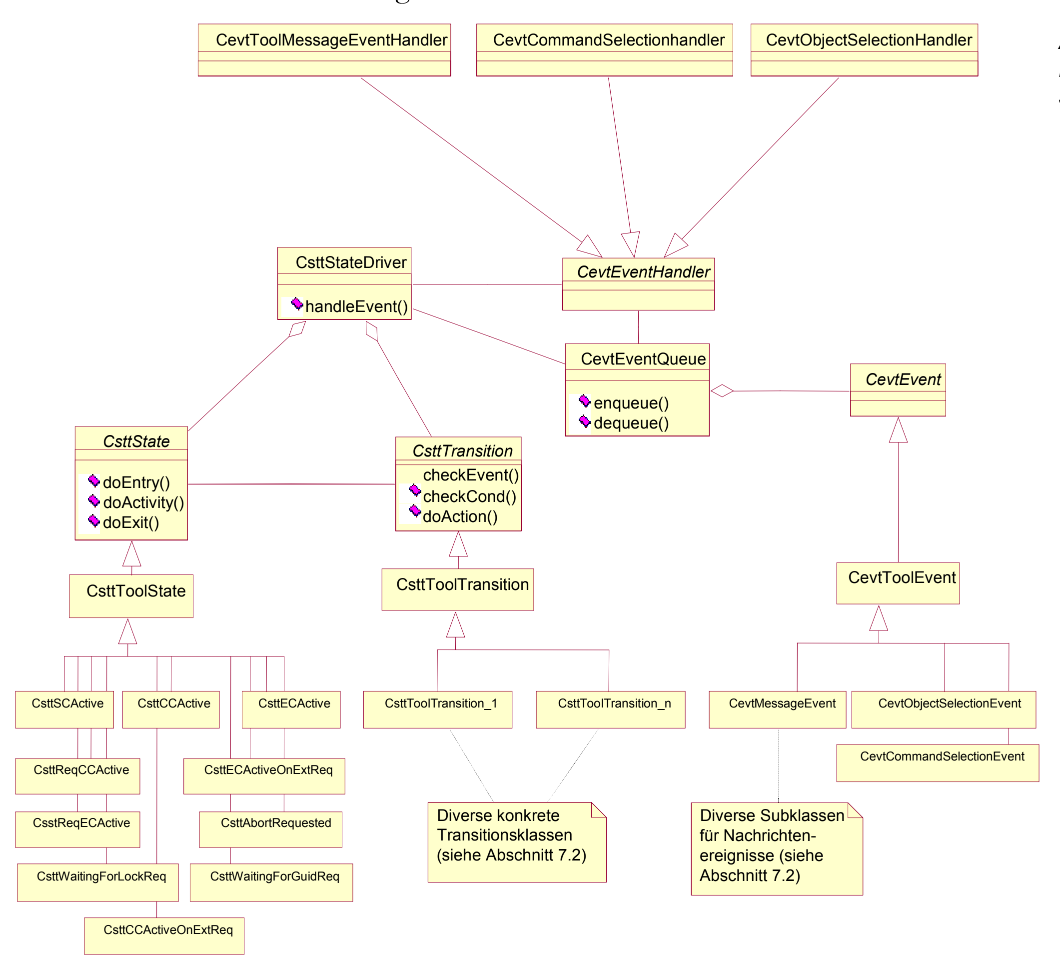


***Abb. 51**: Detailstruktur des Teilsystems StateManager*

Dynamische Protokollanpassungen zur Laufzeit

Die CsttStateDriver-Klassen gewinnen zusätzliche Flexibilität dadurch, dass die Topologie des Zustands-/Transitionsgraphen auch noch nach der Initialisierung über entsprechende Methoden (Hinzufügen oder Entfernen von CsttState- und CsttTransition-Objekten, Verändern der Verknüpfung zwischen CsttState- und CsttTransition-Objekten) modifiziert werden kann. Damit können Änderungen des globalen Werkzeugverhalten auf sehr einfache Weise und bei Bedarf sogar dynamisch, d.h. zur Laufzeit, adaptiert werden. In [Dömg99] wird beispielsweise eine Erweiterung des Zustandsprotokolls um zusätzliche Interaktionen mit einem Filtermechanismus für Nachvollziehbarkeitsinformationen beschrieben, die auf Architekturebene sehr einfach und an einer Stelle zentralisiert durch Hinzufügen

von Objekten einiger neuer `CsttState`- und `CsttTransition`-Objekte zum `CsttToolStateDriver` umgesetzt werden konnte.

Ereignisverwaltung

Die Funktionsweise der `CsttStateDriver`-Klasse ist ereignisgesteuert, so dass diese Klasse eng mit dem `Event`-Teilsystem (Klassenpräfix `Cevt`) zusammenarbeitet. Ereignisse werden entweder durch den Empfang von Nachrichten aus der Leitdomäne im Teilsystem `MessageInterface` oder durch Benutzerinteraktionen (Objekt- und Kommandoselektionen) im Teilsystem `GenericGUI` ausgelöst. Diese Ereignisse werden durch entsprechende EventHandler-Objekte (`CevtMessage-EventHandler`, `CevtCommandSelectionHandler`, `CevtObjectSelectionHandler`) entgegengenommen, in eine einheitliche Form transformiert (als Objekte von Subklassen der Basisklasse CevtEvent) und in eine Ereignis-Warteschlange (`CevtE-ventQueue`) eingereiht. Sobald ein Ereignis im `CevtEventQueue`-Objekt vorliegt, wird die Ereignisbehandlung an die Methode `handleEvent` des `CsttStateDriver`-Objekts delegiert. Ausgehend vom aktuell aktiven Zustand werden alle adjazenten Transitionen mithilfe der `CsttTransition`-Methoden `checkEvent` und `checkCondi-tion` daraufhin geprüft, ob sie schalten können. Falls eine schaltbare Transition gefunden wurde, wird ein Zustandsübergang vollzogen, indem nacheinander die `doExit`-Methode des aktuellen `CsttState`-Objekts, die `doAction`-Methode der schaltenden Transition sowie die `doEntry`- und `doActivity`-Methode des Zielzustands aufgerufen werden. Dieser Vorgang iteriert, solange weitere Ereignisse im `CevtEventQueue` vorliegen. In der gängigen Framework-Terminologie fungiert somit die `handleEvent`-Methode als *Template-Methode*, während die `doX`-Methoden *Einschubmethoden* darstellen, wobei für letztere zur Laufzeit unter Ausnutzung der Polymorphie jeweils die Ausprägungen der spezifischen Subklassen aufgerufen werden.

Zusammenfassend können wir festhalten, dass die konzeptuelle Spezifikation des Werkzeugverhaltens aus Abschnitt 7.2.1 innerhalb des GARPIT-Frameworks als eigener Architekturbaustein (Teilsystem `StateManager`) realisiert wurde. Aus architektureller Perspektive hat diese Entwurfsentscheidung eine Reihe signifikanter Vorteile:

- *Kapselung der Werkzeugkontrolle*: Alle Dynamikaspekte des Interaktionsprotokolls mit der Leitdomäne und der Reaktion auf Benutzerereignisse sind im `StateManager`-Teilsystem gekapselt. Änderungen der Verhaltensspezifikation wirken sich nur lokal auf dieses Teilsystem aus, andere Teilsysteme sind nicht betroffen.
- *Flexible Erweiterbarkeit und Anpassbarkeit*: Die explizite Übertragung eines Zustandsdiagramms in eine Klassenstruktur ähnelt auf den ersten Blick dem in [GHJV95] vorgeschlagenen Entwurfsmuster *State*, das eine einfache Variierbarkeit des zustandsbasierten Verhaltens von Objekten zum Ziel hat. Im Gegensatz zum *State*-Muster werden bei uns jedoch Transitionen nicht in Codefragmenten von State-Objekten hart kodiert, sondern ebenfalls explizit repräsentiert. Zwar benötigen wir dann für jeden Transitionstypen eine zusätzliche Klasse, gewinnen dafür aber erheblich an Flexibilität. So kann die im `CsttStateDriver`-Objekt verwaltete Verknüpfung zwischen Zustands- und Transitionsobjekten sogar dynamisch verändert werden, um z.B. Protokolländerungen zur Laufzeit zu realisieren. Diese Eigenschaft wurde u.a. bei der Einbindung eines Filtermechanismus für die

Aufzeichnung von Nachvollziehbarkeitsinformationen ausgenutzt [Dömg99].

- *Entkopplung zwischen High-Level- und Low-Level-Ereignisverwaltung*: Über die Klasse CevtEventQueue wird eine High-Level-Ereignisverwaltung realisiert, die von den Low-Level-Ereignisschleifen in externen Subsystemen (Nachrichtenschnittstellen, Benutzeroberfläche) weitgehend entkoppelt ist. Dadurch kann beispielsweise im Zustand CsttWaitingForLockRequest ein *logisch* synchrones Warten auf eine Antwortnachricht aus der Leitdomäne emuliert werden, obwohl die physischen Ereignisse in Wirklichkeit weiterhin asynchron eintreffen. Dies verhindert insbesondere, dass Low-Level-Ereignisse in der grafischen Benutzeroberfläche, die für die globale Werkzeugkontrolle irrelevant sind, blockiert werden (z.B. Aktualisierung der Fensterdarstellung nach Verändern der Fenstergröße, Verschieben des Fensters etc.).
- *Wiederverwendung in der Leitdomäne*: Das von den Klassen CsttStateDriver, CsttState, CsttTransition und CevtEventqueue gebildete Grundgerüst konnte in der Leitdomäne zur Realisierung der globalen Prozessmaschinenkontrolle unverändert wiederverwendet werden.

7.3.2.4 MessageInterface

Das Teilsystem MessageInterface realisiert eine Nachrichten-basierte Schnittstelle, über die das Werkzeug Nachrichten an die Prozessmaschine verschicken und von dort empfangen kann (siehe Abb. 52). Dieses Paket besteht aus den beiden Teilpaketen CommunicationChannel (Klassenpräfix Cskt[33]) und Message (Klassenpräfix Cmsg).

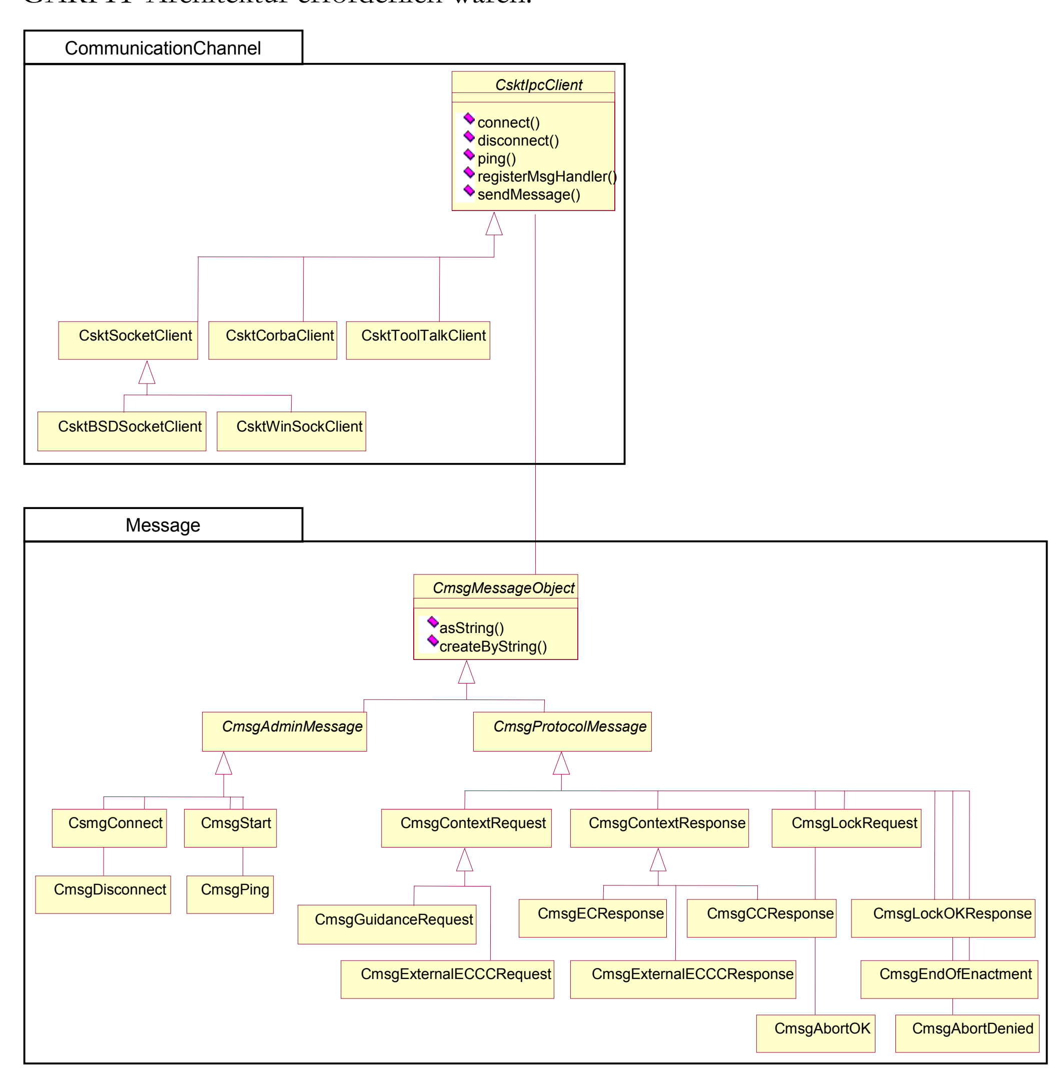
CommunicationChannel : Abstraktion vom zugrunde liegenden Mechanismus zur Interprozess-kommunikation

Das Teilsystem CommunicationChannel realisiert den Kommunikationskanal zum Kommunikationsmanager. Die Schnittstelle des CommunicationChannel gliedert sich in zwei Teilbereiche. Zum einen werden administrative Methoden für die An- und Abmeldung vom Kommunikationsmanager sowie für die Erkennung von Verbindungsabbrüchen und deren eventuelle Wiederherstellung angeboten. Zum anderen exportiert der CommunicationChannel Methoden zum synchronen und asynchronen Versand von Nachrichtenobjekten sowie zur Registrierung eines Objekts der Klasse CevtMessageEventHandler, über das der asynchrone Empfang von Nachrichten vom Kommunikationsmanager an den StateManager gemeldet wird (siehe Abschnitt 7.3.2.3). Der CommunicationChannel abstrahiert vollständig vom zugrunde liegenden Mechanismus zur Interprozesskommunikation. Im Laufe der Entwicklung des PRIME-Frameworks haben wir zur Realisierung des CommunicationChannels zunächst BSD Sockets und Sun's ToolTalk unter Unix, später dann WinSock sowie CORBA unter Windows NT/9x verwendet. Vom Austausch des Kommunikationsmechanismus war lediglich der Implementierungsteil des CommunicationChannel-Pakets betroffen, während die externen Schnittstellen

[33] Das Klassenpräfix Cskt rührt von der ursprünglichen Paketbezeichnung Socket. In den ersten Versionen des PRIME-Frameworks basierte die Interprozess-Kommunikation ausschließlich auf BSD Sockets. Mittlerweise haben wir jedoch auch andere Kommunikationsmechanismen (ToolTalk und CORBA) als Implementierungsplattform eingesetzt, so dass wir nun die technologieneutrale Bezeichnung CommunicationChannel bevorzugen.

erhalten blieben, so dass keinerlei Änderungen an den übrigen Teilsystemen der GARPIT-Architektur erforderlich waren.

***Abb. 52**: Detailstruktur des Teilsystems MessageInterface*



Message: abstrakte Datentypen für Nachrichtenobjekte

Die Nachrichtentypen, die zwischen einem Werkzeug und dem Kommunikationsmanager ausgetauscht werden, sind im Teilsystem `Message` zusammengefasst. Die Klasse `CmsgMessageObject` bildet die Basisklasse, von der alle weiteren Nachrichtentypen abgeleitet werden. Die wichtigste Eigenschaft aller Nachrichtenobjekte besteht darin, dass sie sich selbst in eine Zeichenkette serialisieren können bzw. aus einer Zeichenkette rekonstruieren können. Dazu definiert die Klasse `CmsgMessageObject` die beiden Methoden `asString` und `createByString`, die von jeder konkreten Nachrichtenklasse zu implementieren sind. Mithilfe dieser Methoden ist der `CommunicationChannel` in der Lage, Nachrichten als Zeichenkette zu verschicken bzw. eine als Zeichenkette erhaltene Nachricht in ein Nachrichtenobjekt umzuwandeln, ohne die Semantik und Struktur der ausgetauschten Nachrichten kennen zu müssen.

Die konkreten Nachrichtentypen gliedern sich in eine Gruppe administrativer Nachrichten (`CmsgConnect`, `CmsgDisconnet`, `CmsgPing`, `CmsgStart` und weitere) und in solche Nachrichtentypen, die im Rahmen des Interaktionsprotokolls aus Abschnitt 7.2 ausgetauscht werden. Wichtig sind hier insbesondere die `CmsgContextRequest`- und `CmsgContextResponse`-Klassen, die eine Kontextanforderung bzw. das Resultat einer Kontextausführung verkapseln. Diese Nachrichtentypen

sind generisch in dem Sinne, dass sie durch ein Kontextdeskriptor-Objekt (CcxtContextDesc) und eine Situationsinstanz-Objekt (CcxtSituationInstance) parametriert werden. Diese Klassen dienen der Laufzeitrepräsentation der im Prozessrepository abgelegten Werkzeugkontexte und -situationen und können sich ebenfalls selbstständig in eine Stringdarstellung umwandeln bzw. daraus konstruieren. Ein Werkzeug kann daher auf Modellierungsebene um weitere Kontext- und Situationstypen erweitert werden, ohne dass es manuell (oder auch generativ, etwa im Sinne der Kompilierung einer IDL-Spezifikation) um neue Nachrichtentypen ergänzt werden müsste. Dies ist ein entscheidender Vorteil gegenüber Object-Brokern wie CORBA oder COM, MessageServer-basierten Ansätzen wie Softbench, ToolTalk oder auch dem GTSL-Ansatz von Emmerich [Emme95], wo für jede neu exportierte Werkzeugfunktion – in unserer Terminologie wäre dies ein Kontext – ein neuer Nachrichtentyp (bzw. Methoden-Skeletons und -Stubs) sowohl auf Werkzeug- als auch auf Klientenseite realisiert werden muss.

Das Teilsystem MessageInterface benötigt keinerlei Adressateninformationen. Alle Nachrichten werden an den Kommunikationsmanager geschickt, der die für die Nachrichtenverteilung erforderlichen Informationen aus dem Umgebungsmodell und der internen Tabelle der laufenden Werkzeug- und Prozessmaschineninstanzen entnimmt und für die korrekte Weiterleitung der Nachrichten an den richtigen Adressaten verantwortlich ist. Die dadurch gewonnene Verteilungstransparenz vereinfacht sowohl den Entwurf des MessageInterface-Teilsystems als auch dessen Nutzung in höheren Architekturschichten erheblich.

7.3.2.5 ContextManager

Das Teilsystem ContextManager ist für die Interpretation der werkzeugrelevanten Anteile des Umgebungsmodells zuständig. Gemäß den Anforderungen aus Abschnitt 7.3.1.1 bestehen die Aufgaben des ContextManagers in der effizienten Laufzeitverwaltung der für ein Werkzeug gültigen Kontextdefinitionen, der prozessmodellkonformen Steuerung von Ausführungs- und Entscheidungskontexten sowie der Erkennung von Kontextaktivierungen durch den Benutzer. Diese drei Aufgaben werden von den Hauptklassen CcxtContextCache, CcxtContextExecutor und CcxtContextMatcher realisiert. Dabei erfolgt die Interaktion des ContextManager-Teilsystems mit der werkzeugspezifischen Basisfunktionalität (Teilsystem T_Actions) und Benutzeroberfläche (T_GUI) über die Verzeichnisklassen des Teilsystem Map (siehe Abb. 53).

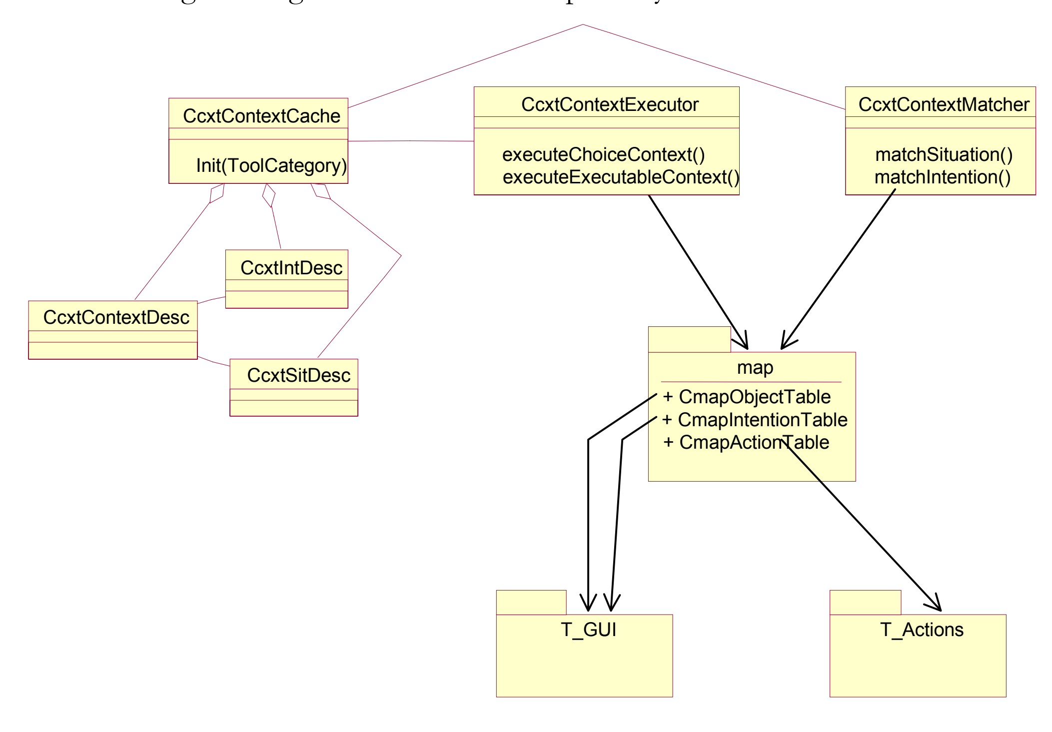
ContextCache

ContextCache: Laufzeitverwaltung der relevanten Kontextdefinitionen

Der ContextCache wird als Objekt der Klasse CcxtContextCache beim Werkzeugstart erzeugt, lädt bei seiner Initialisierung die benötigten Kontext-, Intentions- und Situationsinformationen aus dem Prozessrepository und repräsentiert diese mithilfe entsprechender Deskriptorklassen (CcxtContextDecs, CcxtSituationDesc, CcxtIntentionDesc etc.). Der Konstruktor der CcxtContextCache-Klasse wird dabei mit der ID der betreffenden Werkzeugkategorie parametriert, so dass nur die für ein Werkzeug relevanten Kontextdefinitionen in den ContextCache geladen werden. Relevant für ein Werkzeug sind alle Ausführungskontexte und Entscheidungskontexte (inklusive Situationen und Intentionen), die im Umgebungsmodell der betreffenden Werkzeugkategorie direkt zugeordnet wurden, sowie alle Kontexte, die im Umgebungsmodell als Alternativen eines werkzeugeigenen Entschei-

dungskontexts definiert wurden. Die Klassen CcxtContextExecutor und CcxtContextMatcher schlagen Kontextinformationen im ContextCache nach und vermeiden so ständige Anfragen an das Prozess-Repository.

***Abb. 53**: Klassenstruktur des Teilsystems ContextManager*



ContextExecutor

ContextExecutor: Steuerung der Kontextausführung

Die Klasse CcxtContextExecutor realisiert die eigentliche Steuerung der prozessmodellkonformen Kontextausführung. Der ContextExecutor wird vom StateManager aufgerufen, sobald ein dem Werkzeug zugeordneter Kontext aktiviert wurde (entweder von außen durch die Prozessmaschine oder durch den ContextMatcher, s.u.). Der ContextExecutor exportiert im Wesentlichen die beiden Methoden executeChoiceContext(CcxtCCDesc, CcxtSituationInstance) und executeExecutableContext(CcxtECDesc, CcxtSituationInstance) für die Aktivierung eines instanziierten Entscheidungs- bzw. Ausführungskontexts.

Aktivierung eines Ausführungskontexts

Bei der Aktivierung eines Ausführungskontexts (Aufruf der Methode executeExecutableContext) ermittelt der ContextExecutor die ID der Aktion, die gemäß Prozessmodell den Ausführungskontext operationalisiert. Der ContextCache delegiert den Aktionsaufruf an die Verzeichnisklasse ActionTable weiter, in der alle Aktionsobjekte unter ihrer Prozessmodell-ID registriert sind. Aufgrund dieser Indirektion benötigt der ContextExecutor keine Kenntnis der werkzeugspezifischen Action-Klassen und kann somit als vollkommen generische Komponente realisiert werden.

Aktivierung eines Entscheidungskontexts

Bei der Aktivierung eines Entscheidungskontexts (Aufruf der executeChoiceContext-Methode) muss der ContextExecutor die Benutzeroberfläche so anpassen, dass nur noch die dem Entscheidungskontext zugeordneten Alternativen vom

Benutzer ausgewählt werden können und die zur aktuellen Situationsinstanz gehörenden Produktteile hervorgehoben werden. Analog zur Ausführung von Aktionen operiert der ContextExecutor nicht direkt auf werkzeugspezifischen Benutzeroberflächen-Objekten, sondern über die Klassen CmapObjectTable und CmapIntentionTable, die eine Indirektion zwischen dem generischen Teilsystem ContextManager und dem werkzeugspezifischen Teilsystem T_GUI herstellen. Dadurch können die ContextManager-Klassen ausschließlich über Prozessmodell-IDs auf Produkten und Kommandoelementen arbeiten, ohne die werkzeugspezifischen Klassen kennen zu müssen.

Die IntentionTable repräsentiert die Menge aller Kommandoelemente eines Werkzeugs und besteht aus Einträgen der folgenden Struktur.

ID der Intention (Prozessmodell)	Zeiger auf assoziierte Kommandoelemente im Teilsystem T_GUI	Aktivierbarkeitsstatus (aktivierbar, deaktiviert)

***Tab. 12**: Struktur der IntentionTable*

Die IntentionTable-Einträge (Instanzen der Klassen CmapCommandDesc) assoziieren also die Prozessmodell-ID einer Intention mit den im Umgebungsmodell definierten Kommandoelement-Objekten (Menüpunkte, Toolbar-Icons, Shortkey-Bindings) und dem aktuellen Aktivierbarkeitsstatus. Der ContextExecutor aktiviert mithilfe der IntentionTable-Methode enableIntention(IntentionID) alle Intentionen, die zu Alternativen des aktuellen Entscheidungskontexts gehören, und deaktiviert mithilfe der Methode disableIntention(IntentionID) alle anderen. Die Änderung des Aktivierungsstatus wird von der IntentionTable an die eigentlichen (werkzeugspezifischen) Kommandoelement-Objekte gemäß dem *Observer*-Entwurfsmuster propagiert.

Auf ähnliche Weise erfolgt die Anpassung der Produktregion eines Werkzeugs. Hier fungiert die ObjectTable als Schnittstelle zwischen dem ContextExecutor und den werkzeugspezifischen Benutzeroberflächenobjekten des Teilsystems T_GUI. Die ObjectTable-Einträge (Objekte der Klasse CmapObjectDesc) weisen im Vergleich zur IntentionTable eine etwas kompliziertere Struktur auf:

ID der Produkt-instanz	ID des Produkttypen der Produktinstanz	Zeiger auf Präsentationsobjekt in T_GUI	Zeiger auf Produktmodellobjekt in T_Model	Darstellungsart (hervorgehoben, selektierbar, nicht selektierbar)	Selektionsstatus (selektiert, nicht selektiert)

***Tab. 13**: Struktur der ObjectTable*

Die ID einer Produktmodellinstanz wird hier mit ihrem Produkttypen sowie mit den korrespondierenden Laufzeit-Objekten aus den Teilsystemen T_Model (objektorientierte Zugriffsschicht) und T_GUI (Benutzeroberfläche) assoziiert. Zudem ist für jede Produktinstanz die aktuelle Darstellungsart sowie ihr Selektionsstatus vermerkt. Bei der Ausführung eines Entscheidungskontexts passt der ContextExecutor die Produktregion des Werkzeugs wie folgt an (siehe auch entsprechende Festlegungen im Umgebungsmodell, Abschnitt 5.5):

- Für alle Produktinstanzen, die zur aktuellen Situationsinstanz gehören, wird die Darstellungsart „hervorgehoben" gesetzt (mithilfe der ObjectTable-Methode setDisplayMode(ProductID, emphasized);
- Für alle *potenziell auswählbaren* Produktinstanzen wird die Darstellungsart „selektierbar" gesetzt. Potenziell auswählbar sind alle Produktinstanzen, deren Produkttyp zur Situation einer Alternative des aktuellen Entscheidungskontexts gehört. Entsprechend ermittelt der ContextExecutor zunächst alle im aktuellen Entscheidungskontext in Frage kommenden Pro-

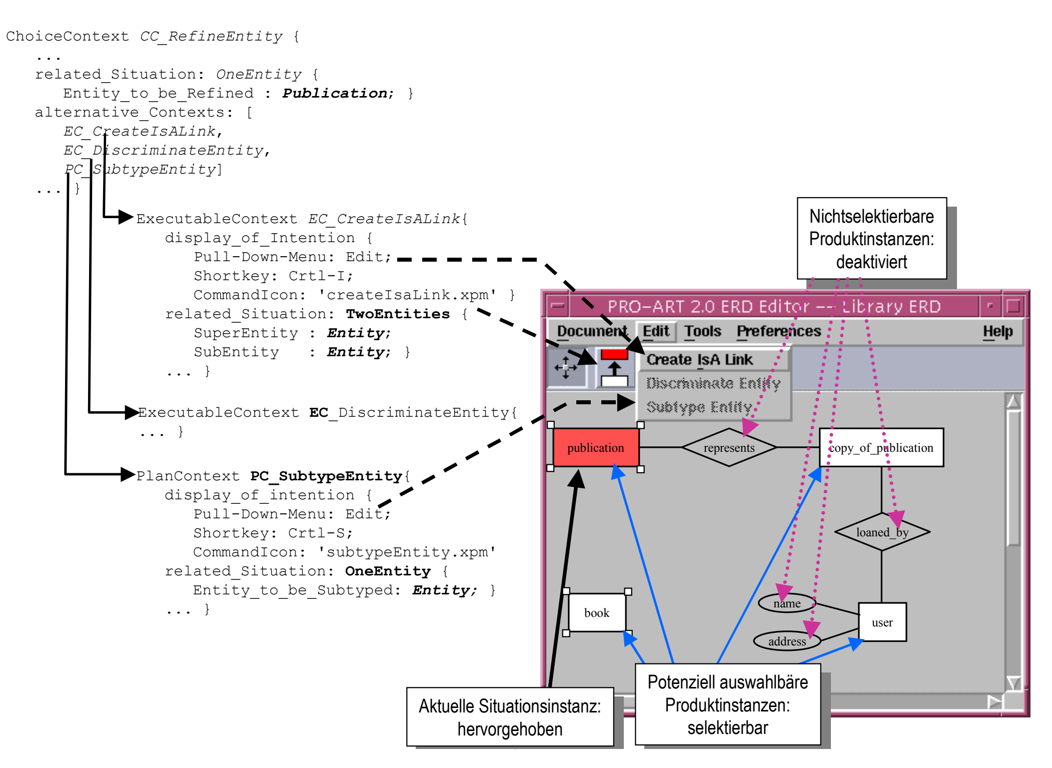
dukttypen und ruft für diese die ObjectTable-Methode setDisplayMode(ProductTypeID, selectable) auf.

- Für alle anderen Produktinstanzen wird die Darstellungsart „nicht selektierbar“ gesetzt.

Wie bei der Anpassung der Kommandoregion, operiert der ContextExecutor nur mit Identifikatoren des Umgebungsmodells. Die Änderung der Darstellungsart der Produktinstanzen durch den ContextExecutor wird von der ObjectTable an die werkzeugspezifischen T_GUI-Objekte propagiert.

Anhand eines Beispiels wollen wir nun die Arbeitsweise des ContextExecutors bei der Ausführung eines Entscheidungskontexts illustrieren. Abb. 54 zeigt den mithilfe des GARPIT-Frameworks entwickelten ER-Editor bei der Ausführung des Entscheidungskontexts CC_RefineEntity. Auf der linken Seite ist in Pseudocode-Notation der Ausschnitt aus der ContextCache-Laufzeitstruktur für die Prozessmodelldefinition des Entscheidungskontexts CC_RefineEntity angegeben. Für die aktuell aktive Kontextinstanz gibt die Laufzeitstruktur an, dass die aktuelle Situationsinstanz an das Produkt Publication (vom Produkttypen Entity) gebunden ist.

***Abb. 54**:*
Anpassung der Werkzeugoberfläche durch ContextExecutor



Gemäß der Definition des Entscheidungskontexts stehen dem Benutzer für die Verfeinerung einer Entität drei alternative Vorgehensweisen zur Auswahl:

- Hinzufügen einer Spezialisierungsbeziehung zu einer existierenden Entität (EC_CreateIsALink);
- Hinzufügen eines Diskriminator-Attributes (EC_AddDiscriminator);
- Erzeugen neuer Entitätstypen als Subtypen von Publication (PC_SubtypeEntity).

Hierbei entsprechen die ersten beiden Alternativen atomaren, werkzeugeigenen Diensten (Ausführungskontexte), während die dritte Alternative ein komplexes

Prozessfragment darstellt (Plankontext). Gemäß den Festlegungen im Umgebungsmodell werden vom ContextManager über die IntentionTable diese Alternativen aktiviert, d.h. in das Edit-Menü eingetragen, durch entsprechende Icons in der Toolbar dargestellt und an die definierten Shortkeys gebunden. Alle anderen Kommandoelemente werden temporär ausgeschaltet und sind nicht zugreifbar (Aktivierbarkeitsstatus „deaktiviert“). In der Produktregion wird als aktuelle Situationsinstanz die Entität Publication in der Darstellungsart hervorgehoben (hier: rot) dargestellt. Die mit den Alternativen assoziierten Situationstypen definieren die Produkttypen, deren Instanzen potenziell ausgewählt werden können. In diesem Fall kommen nur noch Entitäten (als Bestandteile der möglichen Situationen OneEntity und TwoEntities) in Frage. Diese werden als selektierbar dargestellt (hier: weiß). Dagegen setzt der ContextExecutor die Darstellungsart von Attributen und Beziehungstypen auf nicht selektierbar (hier: grau), da diese aufgrund der Prozessdefinition nicht zu gültigen Situationen der aktuell möglichen Alternativkontexte beitragen können.

ContextMatcher

Die Aufgabe des ContextMatchers besteht im Abgleich der Benutzerinteraktionen (Kommando- und Produktauswahlen) mit den Definitionen der zum aktuellen Entscheidungskontext gehörenden Alternativen. Sobald die aktuelle Kommando- und Produktauswahl einem im aktuellen Entscheidungskontext erlaubten Alternativkontext entspricht, wird ein ContextMatched-Ereignis an den StateManager geschickt.

Entsprechend der Unterteilung einer Kontextdefinition in einen Intentions- und Situationsteil muss der ContextMatcher die Aktivierung von Intentionen einerseits und Situationen andererseits erkennen. Wie beim ContextExecutor werden durch Benutzerinteraktionen ausgelöste Zustandsänderungen von Kommandoelementen und Produktinstanzen über die IntentionTable und die ObjectTable zwischen den Teilsystemen ContextManager und T_GUI kommuniziert.

Die Intentionserkennung ist trivial, da es zwischen einem aktivierten Kommandoelement und der entsprechenden Intention eine direkte Zuordnung gibt. Etwas schwieriger ist die Situationserkennung, da sich eine Situation in der Regel aus mehreren Produktinstanzen bestimmter Produkttypen zusammensetzt.

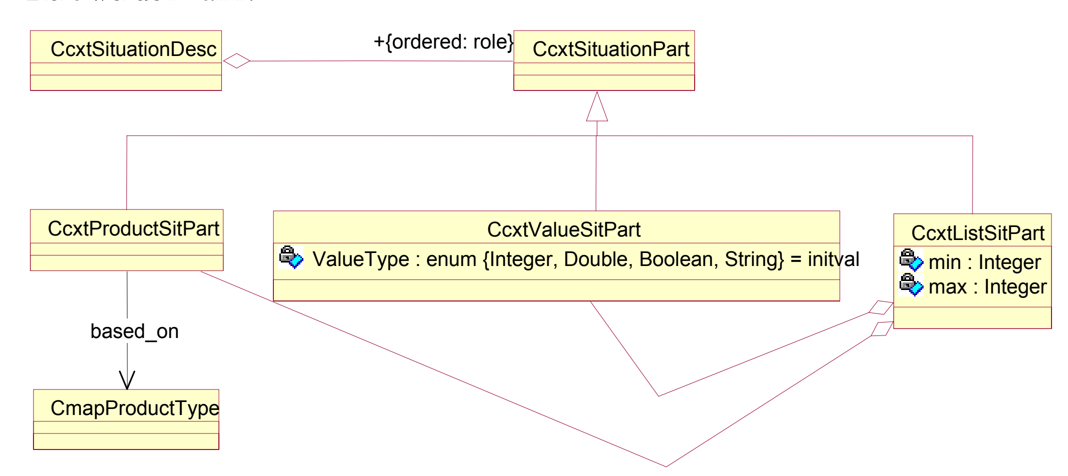
Situationssprache

Aus Werkzeugsicht definiert eine Situation eine Konstellation von Produkten, die vom Benutzer in der Benutzeroberfläche des Werkzeugs durch Selektion, Markierung o.ä. als *aktiv* ausgewiesen wurden. Um solche Produktkonstellationen charakterisieren zu können, haben wir als Verfeinerung des Situationsbegriffs des NATURE-Prozessmodells eine einfache Situationssprache entwickelt, deren Metamodell in Abb. 55 repräsentiert ist. Die Beschreibung eines Situationstypen (zur Laufzeit repräsentiert durch ein Objekt der Klasse CcxtSituationDesc[34]) besteht aus einem oder mehreren getypten Situationsteilen (CcxtSituationPart). Die

[34] Situationstyp-Beschreibungen werden im Prozessrepository persistent abgelegt. Aus diesen Beschreibungen werden Laufzeit-Objektstrukturen der korrespondieren Klassen im Teilsystem ContextManager (Paketkürzel Ccxt) kreiert.

einzelnen Situationsteile eines Situationstypen sind geordnet und können über einen Rollennamen (Attribut `role` der Aggregation zwischen `CcxtSituationDesc` und `CcxtSituationPart`) referenziert werden. Situationsteile werden nach der Art ihrer Typisierungen weiter differenziert (im Metamodell dargestellt durch die Subklassen `CcxtProductSitPart`, `CcxtValueSitPart` und `CcxtListSitPart`). Im Normalfall referenzieren Situationsteile Produkttypen des zugrunde liegenden Produktmodells (`CcxtProductSitPart`). Weiterhin sind Basistypen wie `integer`, `double`, `string` oder `boolean` (`CcxtValueSitPart`) möglich sowie listenwertige Ausprägungen der vorgenannten Typen (`CcxtListSitPart`), wobei mithilfe der Attribute `min` und `max` die Kardinalität eines listenwertigen Situationsteils spezifiziert werden kann.

Abb. 55: Metamodell für Situationstyp-Spezifikation



Der Methodenmodellierer kann neue Situationstypen entweder direkt mithilfe des NATPROC-Editors (siehe Abschnitt 7.1.2.1) in das Prozess-Repository einpflegen oder auch textuell definieren. Textuelle Situationsspezifikationen werden dann von einem Situationscompiler in entsprechende Einträge im Prozess-Repository übertragen und müssen den folgenden in EBNF notierten Syntaxregeln genügen:

Abb. 56: Syntax für Situationstyp-Spezifikationen

```
sit_spec      ::= situation identifier with (sit_part)+ end
sit_part      ::= identifier : sit_part_type;
sit_part_type ::= (product identifier |
                   list [num_literal, (num_literal | *),
                        sit_part_type] |
                   value value_type);
value_type ::= (integer | double | string | boolean)
```

Abb. 57 illustriert die Spezifikation von Situationen am Beispiel von drei Situationstypen, die für Kontexte des ER-Editors definiert wurden.

Abb. 57: Beispiele für Situationstypen

```
situation TwoEntities with
    sub_entity   : product Entity;
    super_entity: product Entity;
end

situation AttributeNameForEntity with
    entity       : product Entity;
    attrname     : value string ;
end

situation AtLeastOneERObject with
    erobjects    : list [1, *, product ER_Object]
end
```

Die Situation TwoEntities ist gegeben, wenn der Benutzer zwei Objekte vom Typ Entity aktiviert hat, wobei das eine Objekt die Rolle subentity und das andere die Rolle superentity einnimmt. Dieser Situationstyp ist Teil des Kontexts Create-IsALink, mit dem eine Spezialisierungsbeziehung zwischen zwei Entitäten erzeugt werden. In der Situation AttributeNameForEntity hat der Benutzer eine Entität (entity) ausgewählt und einen String (attrname) vorgegeben. Diese Situation ist Teil des Kontexts CreateAttribute, mit dem ein neues Attribut mit Namen attrname zur Entität entity hinzugefügt werden kann. Die Situation AtLeast-OneERObject ist dann gültig, wenn der Benutzer mindest ein bis beliebig viele ER-Objekte ausgewählt hat. Der Produkttyp ER_Object ist hierbei eine Oberklasse für alle einem ER-Diagramm vorkommenden Konstrukte (Entität, Beziehung, Attribut, Spezialisierungsbeziehungen etc.).

Beschränkung der Situationsspezifikation auf strukturellen Aufbau

Unsere Situationssprache erlaubt in der aktuellen Implementierung nur die Spezifikation des *strukturellen* Situationsaufbaus auf der Ebene von Produkttypen. Insbesondere ist keine Formulierung von zusätzlichen Bedingungen unter Verwendung von Attributen und Methoden des zugrunde liegenden Produktmodells möglich. Diese Beschränkung hat im Wesentlichen den folgenden Grund. Der Zugriff auf Attribute und Methoden des zugrunde liegenden Produktmodells würde naheliegenderweise über die für diesen Zweck vorgesehene objektorientierte Zugriffsschicht des werkzeugspezifischen Produktmodell T_Model erfolgen. In diesem Fall müsste der ContextMatcher jedoch die spezifischen Schnittstellen der T_Model-Klassen kennen, um bei der Situationsauswertung auf die entsprechenden Attribute und Methoden zugreifen zu können. Bei der von uns gewählten Implementierungssprache C++ würde dies in einem *statischen* Bezug zwischen dem ContextMatcher und den T_Model-Klassen resultieren. Dies würde jedoch mit unserer Intention, den ContextMatcher als generische Framework-Komponente zu realisieren, konfligieren. Außerdem würde jede neue Situationsspezifikation die Generierung (oder manuelle Implementierung) und Rekompilierung entsprechenden Auswertungscodes nach sich ziehen, was im Widerspruch zur einfachen Anpassbarkeit an neue Prozessdefinitionen steht.

Dynamische Bindung von Bedingungsausdrücken an C++-Attribute und -Methoden schwierig

Ein Ausweg könnte darin bestehen, in einer generischen Auswertungskomponente Attribut- und Methodennamen, die in den externen Situationsspezifikation referenziert werden, *dynamisch* an die entsprechenden T_Model-Attribute und -Methoden zu binden. In C++ ist dies jedoch nur mit erheblichem Zusatzaufwand und auf sehr ineffiziente Weise möglich, da dies letztendlich in einer kompletten Abbildung der statischen Klassenstruktur in Laufzeit-Typobjekte resultieren würde. Wesentlich geeigneter wären hier Implementierungsplattformen, die Mechanismen zur dynamischen Introspektion und Reflexion von Haus aus anbieten, wie z.B. Java, Tycoon [Matt93] oder das Dynamic Invocation Interface von CORBA. Diese Technologien waren jedoch zum Zeitpunkt der Erstellung des PRIME-Frameworks nicht verfügbar oder schieden aus anderen Gründen aus (siehe auch Abschnitt 7.3.3 „Implementierung“). Für den Verzicht auf die Spezifikation zusätzlicher Situationsbedingungen waren also weniger konzeptionelle als vielmehr implementatorische Gründe ausschlaggebend. Es hat sich jedoch bei unseren Beispielanwendungen herausgestellt, dass der Verzicht auf die Referenzierung von Attributen und Methoden in der Praxis keine große Rolle spielt.

Matching-Algorithmus

Für die Situationserkennung müssen die Situationstypen der Alternativen des aktuell aktiven Entscheidungskontexts *EK* mit der Menge *P* der selektierten Produktinstanzen in der Produktregion eines Werkzeugs abgeglichen werden. Bei einer erfolgreichen Situationserkennung wird einer (oder auch mehrere) der in Frage kommenden Situationstypen durch eine passende Konstellation von Produktinstanzen instanziiert. Der Matching-Algorithmus hat folgende Struktur:

***Abb. 58**: Grundstruktur des Matchingalgorithmus*

```
M := ∅;
forall S (S ist Situationstyp einer Alternative von EK) do
  M := M ∪ BindeSituation( S, P);
end
falls M=∅: Situationserkennung nicht erfolgreich
falls |M|>1: wähle eine Situationsinstanz
```

Im Folgenden betrachten wir den Teilalgorithmus `BindeSituation` genauer. Dieser Algorithmus versucht, einen Situationstypen *S* in einer Menge *P* von Produktinstanzen zu instanziieren.

Damit ein Situationstyp *S* instanziiert werden kann, muss jeder seiner *m* Situationsteile r_i (i = 1,..., m) an einen Wert gebunden sein. Wir sagen in diesem Fall, dass die Situation *S gilt* und dass die Bindung *s* eine *Instanz* von *S* ist. Formal ist *s* ein m-Tupel (s_1,...,s_m), wobei für die einzelnen Situationsinstanzteile s_i (i = 1,.., m) folgendes gelten muss:

- wenn der korrespondierende Situationsteil r_i von der Form

  r_i : `product` T_i

  ist, muss s_i eine Produktinstanz sein und Typ(s_i) $\leq T_i$ (d.h. s_i muss eine Instanz vom Produkttypen T_i oder einem seiner Subtypen sein);

- wenn der korrespondierende Situationsteil r_i von der Form

  r_i : `value` V_i

  ist, muss s_i ein Wert vom Basistypen V_i sein;

- wenn der korrespondierende Situationsteil r_i von der Form

  r_i : `list` [min_i, max_i, `product` T_i]

  ist, muss gelten:

  s_i = [p_1,..., p_k] ist eine Liste von Produktinstanzen mit $Typ(p_j) \leq T_i$ ($j = 1,..,k$) und $min_i \leq k \leq max_i$;

- wenn der korrespondierende Situationsteil r_i von der Form

  r_i : `list` [min_i, max_i, `value` V_i]

  ist, muss gelten:

  s_i = [v_1,..., v_k] ist eine Liste von Werten mit $Typ(v_j) = V_i$ ($j = 1,..,k$) und $min_i \leq k \leq max_i$;

Der nachfolgend skizzierte Matching-Algorithmus erhält als Eingabe einen Situationstypen *S* und eine Menge *P* von aktivierten Produktinstanzen. Der Algorithmus ermittelt, ob *S* gilt und gibt im Falle eines erfolgreichen Matchings eine gültige Situationsinstanz *s* aus. Wir betrachten dabei nur solche Situationstypen, deren sämtliche *m* Situationsteile Listen von Produkttypen referenzieren, die also von der folgender Form sind:

```
situation S with
  r_1 : list[ min_1, max_1, T_1 ];
  ...
  r_m : list[ min_m, max_m, T_m ];
end
```

Dies ist keine Beschränkung der Allgemeinheit, da zum einen ohnehin nur gegenüber einer Menge in der Benutzeroberfläche aktivierter *Produktinstanzen* abgeglichen wird, so dass Wert-basierte Situationsteile (z.B. r_i : `value integer`) durch den Situations-Matcher nicht gebunden werden brauchen. Zum anderen kann ein Produkt-basierter Situationsteil als Spezialfall eines Listen-basierten Situationsteils aufgefasst werden, bei dem die min- und max-Kardinalität jeweils auf 1 gesetzt ist, d.h. folgende Situationsspezifikationen *S* und *S'* sind äquivalent:

```
situation S with                situation S' with
  ...                             ...
  r_k : T_k;                      r_k : list [ 1, 1, T_k ];
  ...                             ...
end                             end
```

Ziel des Algorithmus `BindeSituation` ist es, die *m* Situationsteile r_i von *S* so an Produktinstanzen binden, dass die Typbedingungen und die *min*- und *max*-Kardinalitäten der Situationsteile erfüllt sind und dass jede Produktinstanz genau einem Situationsteil zugeordnet wird.

***Abb. 59**: Algorithmus* `BindeSituation`

Eingabe:
  S: Situationstyp der folgenden Form:
```
    situation S with
      r_1 : list[ min_1, max_1, T_1 ];
      ...
      r_m : list[ min_m, max_m, T_m ];
    end
```
  $P = \{ p_1,...,p_n \}$, Menge von Produktinstanzen

Ausgabe:
  Situationsinstanz $s = (r_1{:}s_1,..., r_m{:}s_n)$, falls Bindung erfolgreich
  `false`, falls keine Bindung möglich

Algorithmus:
  // Vorbereitungsphase
  Sortiere die Situationsteile $[r_1,...,r_m]$ zu $[r_{i_1},..., r_{i_m}]$ so dass gilt:

(1) $\forall\, k,l \in \{1,...,m\}, k<l : T_{i_k} \leq T_{i_l}$ oder T_{i_k} und T_{i_l} stehen nicht in einer Spezialisierungsbeziehung (d.h. Situationsteile mit spezielleren Produkttypen werden nach vorne sortiert.)
(2) $\forall\, k,l \in \{1,...,m\}, k<l : T_k = T_l \Rightarrow i_k < i_l$
(d.h. die Reihenfolge von Situationsteilen mit gleichem Produkttypen bleibt in der Umsortierung erhalten.)

```
// Phase 1: Bedienung der min-Anforderungen jedes Situationsteils
forall r_{i_k} (1 ≤ k ≤ m) do {
    // Liste von Produktinstanzen passenden Typs in P zusammenstellen
    P´ := [ p ∈ P / Typ( p ) ≤ T_{i_k}];
    if |P´| < min_i
            return false
    else {
            // Sortierung gemäß Einordnung in Spezialisierungshierarchie
            Sortiere P´ = [ p_1,...,p_k] so dass gilt:
                    ∀ r,s ∈ {1,.., k}: r<s ⇒ Typ(p_r) ≤ Typ(p_s)
            // der Situationsteilinstanz s_i werden die ersten min_i Instanzen
            // in P´ zugeordnet
            s_{i_k} := P´.firstElements(min_{i_k})
            P := P - P´.firstElements(min_i);
    } // else
} // forall

// Phase 2: Bindung der verbliebenen Produktinstanzen
forall r_{i_k} (1 ≤ k ≤ m) do {
    // Liste von Produktinstanzen passenden Typs in P zusammenstellen
    P´ := [ p ∈ P / Typ( p ) ≤ T_{i_k}];
    // der r_{i_k}-te Situationsteil kann noch maximal
    // max_{i_k} – min_{i_k} Produktinstanzen von T_{i_k} aufnehmen
    if | P´| ≤ max_{i_k} – min_{i_k} {
            s_i.append( P´)
            P := P - P´
    } else {
            s_{i_k}.append( P´.firstElements(max_{i_k} – min_{i_k}))
            P := P - P´. firstElements(max_{i_k} – min_{i_k})
    } // else
} // forall

if P = ∅
    return s =(s_1, ...,s_m)
else
    // es konnten nicht alle Produktinstanzen zugeordnet werden
    return false
```

Der Algorithmus läuft in zwei Phasen ab. Im ersten Durchgang wird versucht, jedem Situationsteil r_i genau min_i Produktinstanzen eines passenden Typs zuzuordnen. Falls dies gelingt, versucht der Algorithmus in der zweiten Phase die verbliebenen Produktinstanzen so auf die Situationsteile r_i aufzuteilen, dass deren max_i-Kardinalität nicht überschritten wird. In beiden Phasen iteriert der Algorithmus je einmal über alle Situationsteile. Da insbesondere in der ersten Phase die Reihenfolge, in der die Zuordnung von Produktinstanzen zu Situationsteilen vorgenommen wird, eine wichtige Rolle spielt, werden die Situationsteile in einer *Vorbereitungsphase* vorsortiert.

Reihenfolge der Situationsteilbindung wichtig!

Hierbei ist zum einen zu berücksichtigen, dass die Produkttypen in einer Spezialisierungshierarchie angeordnet sind und Produktinstanzen eines bestimmten Typs auch zu Situationsteilen zugeordnet werden können, die einen in der Spezialisierungshierarchie allgemeineren Typen verlangen (Substituierbarkeitsprinzip). Daher müssen Situationsteile mit spezielleren Typanforderungen vor solchen mit allgemeineren Typanforderungen an die geforderte *min*-Anzahl von Produktinstanzen gebunden werden. Ein Beispiel aus der Situationsmodellierung des ER-Editors illustriert die potenziell auftretenden Probleme (siehe Abb. 60) bei einer falschen Reihenfolge der Situationsteil-Bindung. Die Situation S_{ER} besteht aus zwei Situationsteilen r_1 und r_2, die jeweils eine Produktinstanz vom Typ `ER_Object` bzw. `ER_Entity` verlangen, wobei `ER_Object` als Wurzel der Produkttyphierarchie des im ER-Editors verwendeten Produktmodells alle anderen ER-Produkttypen subsummiert (`ER_Entity`, `ER_Relationship`, `ER_Attribute`). Die Menge der aktuell gültigen Produktinstanzen besteht aus dem Objekt `book` von Typ `ER_Entity` und `lend_by` vom Typ `ER_Relationship`. Wenn zuerst der speziellere Situationsteils r_2 und dann r_1 gebunden wird, erhalten wir s = (r_1: `lend_by`, r_2: `book`) als gültige Instanz von S. Wird dagegen zunächst der allgemeinere Teil r_1 gebunden, käme entweder `book` oder `lend_by` als zugeordnete Produktinstanz in Frage. Bei der Wahl von book schlägt jedoch die weitere Zuordnung fehl, da dann keine passende Produktinstanz für den spezielleren Situationsteil r_2 mehr übrig ist (siehe Abb. 60).

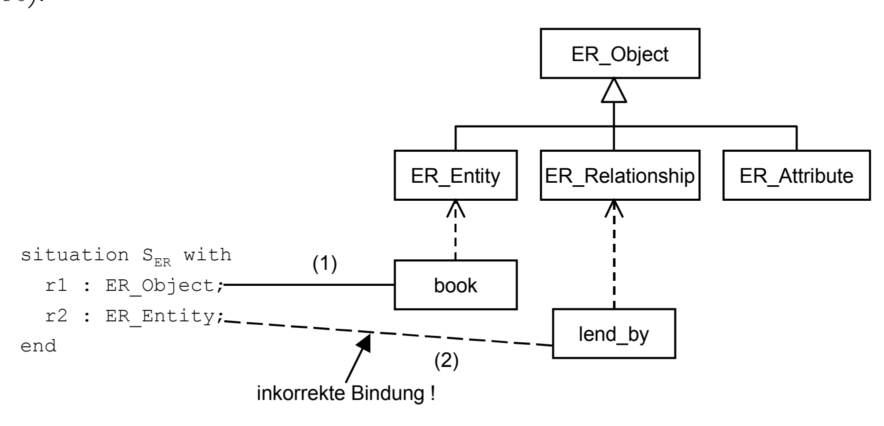


***Abb. 60**:*
Inkorrekte Bindung bei falscher Reihenfolge der Zuordnung

Auflösung von Wahlfreiheiten bei der Zuordnung von Produktinstanzen

Ein weiteres Problem ergibt sich dadurch, das bei Situationsteilen, die den gleichen Typen referenzieren ist, Wahlfreiheiten bei der Zuordnung von Produktinstanzen bestehen. Aus semantischer Sicht sind die Situationsteile jedoch häufig unterschiedlich zu behandeln, was sich in den Rollenbezeichnern ausdrückt. Verdeutlicht wird dies anhand des Situationstypen `TwoEntities` :

```
situation SIsa with
  subEntity : product ER_Entity;
  superEntity : product ER_Entity;
end
```

Diese Situation ist Teil des Kontextes EC_CreateIsaLink, der eine *gerichtete* Spezialisierungsbeziehung zwischen einer Entität in der Rolle subEntity und einer Entität in der Rolle superEntity erzeugt. Falls die Menge der aktuell aktiven Produktinstanzen aus den ER_Entity-Objekten book und publication besteht, gäbe es zwei mögliche Bindungen: s_1 = (subEntity : book, superEntity: publication) und s_2 = (subEntity: publication, superEntity : book), wovon offensichtlich nur s_1 semantisch sinnvoll ist. Eine prinzipielle Möglichkeit zur Auflösung solcher Mehrdeutigkeiten besteht darin, diese explizit vom Benutzer vornehmen zu lassen, etwa über einen spezielles Dialogfenster. Da sich dies jedoch als sehr mühselig erwies, erfolgt in der Realisierung des Situations-Matchers eine automatische Zuordnung nach folgendem Schema. Jede Produktinstanz trägt einen Zeitstempel, der den Zeitpunkt ihrer Aktivierung angibt (die Aktivierung mehrerer Produktinstanzen erfolgt durch akkumulatives Selektieren). Entsprechend der Reihenfolge der Situationsteile in einer Situationsspezifikation werden dann die Produktinstanzen nach ihrem Alter gebunden[35]. Wenn also im obigen Beispiel zuerst book und dann publication selektiert worden wäre, ergäbe sich die Situationsinstanz s_1. Zu beachten ist, dass sich der Benutzer über die Reihenfolge der Situationsteile in einer Situationsspezifikation im Klaren sein muss, damit er die Produktinstanzen in der richtigen Reihenfolge selektiert. In der Praxis erwies sich dies jedoch nur selten als Problem, da in den wenigen Fällen, in denen Mehrdeutigkeiten auftreten konnten, die Reihenfolge intuitiv aus der Semantik des Kontexts hervorging[36].

Aufwandsbetrachtung

Anders als in Abb. 59 dargestellt, ist der Algorithmus in der GARPIT-Realisierung dergestalt optimiert, dass die Situationsteile einer Situation bereits vorsortiert sind (dies muss nur einmal beim Programmstart getan werden) und alle Produktinstanzen zu Beginn des Algorithmus in eine gemäß ihrer Typzugehörigkeit sortierte Liste gebracht werden. Für die Produktinstanzsortierung beträgt der Aufwand mit den üblichen Sortierverfahren $O(n \log(n))$, wobei $n = |P|$ ist. In den beiden Iterationen des Algorithmus entfällt somit die Sortierung von P und es können mit konstantem Aufwand jeweils die Kopfelemente von P entfernt und den entsprechenden Situationsinstanzteilen s_i zugeordnet werden. Da $m \leq n$ gilt (jedem Situationsteil muss mindestens eine Produktinstanz zugeordnet werden), bringen die über m laufenden Iterationen in den beiden Hauptphasen des Algorithmus keine zusätzliche Komplexität mit, d.h. der Aufwand bleibt insgesamt bei $O(n \log(n))$.

---

[35] In absteigender Reihenfolge, d.h. der oberste Situationsteil wird an die älteste Produktinstanz gebunden

[36] Darüber hinaus entspricht dieses Verfahren der gängigen Benutzerführung in den meisten Programmen mit Multiobjekt-Selektion und asymetrischen Operationen (siehe z.B. die *Align*-Funktion in Microsoft Powerpoint).

7.3.3 Implementierung

Das GARPIT-Framework wurde (wie das gesamte PRIME-Framework) komplett in C++ realisiert. Die Wahl der Programmiersprache wurde zum Zeitpunkt der initialen Entwicklung des Frameworks im Wesentlichen durch die hauptsächlich in C++ verfügbaren APIs der verwendeten Fremd-Software (Client-Bibliotheken der Datenbankmanagementsysteme, GUI-Bibliotheken, Kommunikationsbibliotheken, allgemeine Containerbibliotheken) diktiert. Mittlerweile wäre sicherlich auch Java eine überlegenswerte Alternative. Als Systemplattform diente zunächst Unix (Solaris 2.7), später dann Windows NT 4.0. Das Prozessrepository wurde auf Basis der relationalen Datenbankmanagementsysteme Sybase 10, MS SQL Server 7 und MS Access 2000 realisiert. Zur Oberflächenprogrammierung haben wir das portable GUI-Toolkit ILOG Views sowie teilweise die Microsoft Foundation Classes (unter Windows NT) verwendet. Die Interprozesskommunikation wurde in unterschiedlichen Varianten realisiert: zum einen elementar auf Basis von BSD Sockets (unter Unix) bzw. WinSock (unter Windows) und zum anderen mithilfe von Sun's Message-Server-Produkt ToolTalk sowie mithilfe der CORBA-Implementierung omniORB. Tab. 14 gibt einen Überblick über die Pakete des PRIME-Rahmenwerks. Insgesamt umfassen die generischen Anteile des GARPIT-Frameworks ca. 84.000 Zeilen (ausführlich kommentierten) C++-Code.

Tab. 14: Größe der GARPIT-Teilsysteme

Paket	Zweck	Größe (kloc)
kit	Programmieren-per-Vertrag, Assoziations-Templates, Generische Attributbäume	10
msg	Nachrichtenobjekte	8
skt	Kommunikationskanal	10
g_dbi	generische Datenbankklassen	5
g_gui	generische GUI-Klassen (Fenster, Dialoge, Menüsteuerung etc.)	12
g_shape	generische Shape-Klassen	11
cxt	Interpretation des Umgebungsmodells	10
stt	globale Zustandskontrolle	7
evt	Ereignisbehandlung	3
map	Verzeichnisklassen für GUI- und Aktionsobjekte	8
Summe		84

Die werkzeugspezifischen Anteile betragen je nach Werkzeug zwischen 4.000 und 41.000 Codezeilen. Für ein durchschnittliches Werkzeug sind zwischen 8.000 und 15.000 Codezeilen für die Realisierung des werkzeugspezifischen Produktmodells, der Benutzeroberflächenelemente und der Aktionen zu veranschlagen. Damit ergibt sich bei den meisten Werkzeugen ein Wiederverwendungsgrad von mehr als 85 % (siehe Tab. 15 für eine Einzelaufstellung).

***Tab. 15**:*
Größe der Werkzeuge und Wiederverwendungsgrad

Werkzeug	Größe des werkzeugspezifischen Codes (kloc)	Wiederverwendung
ER-Editor	9	90 %
Entscheidungs-Editor	14	85 %
Abhängigkeits-Editor	11	88 %
Hypertext-Editor	9	90 %
Ziel-Editor	18	82 %
MSC-Editor	8	91 %
Whiteboard-Editor	18	82 %
Review-Manager	23	79 %
Produktmodell-Editor	8	91 %
Fließbild-Editor	41	67 %
VeDa-Editor	24	78 %
Task-Manager	8	91 %
Werkzeugmodell-Editor	16	84 %
PC-Editor (SLANG)	5	94 %
PC-Editor (Statecharts)	4	95 %
Anleitungswerkzeug	9	90 %
Prozessspuren-Visualisierer	7	92 %

7.3.4 Beispielanwendung des GARPIT-Frameworks

In diesem Abschnitt illustrieren wir die Arbeitsschritte bei der Entwicklung eines GARPIT-basierten Werkzeugs am Beispiel des ER-Editors. Der ER-Editor ist ein einfaches Werkzeug zur Entity-Relationship-Modellierung, der für die Requirements Traceability-Umgebung PRO-ART [Pohl96] entwickelt wurde.

Die Entwicklung eines Werkzeugs gliedert sich in drei Phasen. Zunächst wird das Werkzeug gemäß der Struktur des Werkzeugmetamodells modelliert, d.h. sein Produktmodell, die Darstellungsarten der Produkttypen und seine Aktionen werden festgelegt. In der zweiten Phase, der Implementierungsphase, werden das Produktmodell, die Benutzeroberfläche und die Aktionen durch Spezialisierung der Framework-Klassen realisiert. In der dritten Phase wird das Werkzeug durch die Definition von Entscheidungs- und Plankontexten in die zu unterstützenden Prozesse eingebunden.

7.3.4.1 Phase 1: Werkzeugmodellierung

Produktmodellierung

Den Ausgangspunkt für die Realisierung eines GARPIT-basierten Werkzeugs bildet die Definition des zugrunde liegenden Produktmodells. Hier unterscheiden wir zwischen einem *Detailmodell,* in dem alle Klassen mit ihren Attributen, Assoziationen und Methoden vollständig spezifiziert sind, und einem *vergröberten Modell*, das nur die Klassen und Spezialisierungsbeziehungen des Detailmodells umfasst. Abb. 61 zeigt das Detailmodell des ER-Editors. Der strukturelle Anteil des Detailmodells wird in ein relationales Datenbankschema übertragen und bildet die Grundlage für die Persistenz des im Teilsystem `ER_Model` implementierten Produktmodells (siehe unten). Das vergröberte Modell wird als Instanz in ein Meta-

schema eingetragen, welches Metainformationen über die vorhandenen Produkttypen und deren Spezialisierungsbeziehungen für den `ContextManager` bereitstellt.

Grafische Symbole, Darstellungsarten und Kommandoelemente

Für die modellierten Produkttypen werden die zu verwendenden grafischen Symbole in den Darstellungsarten *hervorgehoben*, *selektierbar* und *deaktiviert* definiert (vgl. Abschnitt 5.4.2). Beispielweise wird für den Produkttypen `ER_Entity` festgelegt, dass Instanzen dieses Typs im ER-Editor durch ein Rechtecksymbol mit zentriertem Text dargestellt werden soll, wobei das Rechteck je nach Darstellungsart rot, weiß oder grau gefüllt wird. Im Werkzeugmetamodell sind die gängigsten grafischen Symbole und Darstellungsarten vordefiniert. Diese werden vom Framework bereits zur Verfügung gestellt. Benötigt der Werkzeugmodellierer darüber hinausgehende Symbole und Darstellungsarten, muss er diese zum Werkzeugmetamodell hinzufügen und in der Implementierungsphase realisieren. Sowohl das Werkzeugmetamodell als auch das Framework sehen entsprechende Anschluss- und Erweiterungsstellen vor. Im Falle des ER-Editors kommen wir jedoch mit den vordefinierten grafischen Objekten aus.

Zusätzlich zu den grafischen Symbolen für Produkte müssen noch die im ER-Editor verwendeten Kommandoelemente (Menüs, Kommando-Icons, Shortkey-Binding) festgelegt werden. Auch hier bieten das Werkzeugmetamodell und das Framework eine Palette vordefinierter Kommandoelemente für die gängigsten Intentionen (z.B. „Open Diagram“), deren konsistente Verwendung in unterschiedlichen Werkzeugen für ein einheitliches „Look and Feel“ sorgen.

Aktionsmodellierung

Die im Werkzeug modellierten Aktionen definieren die Basisfunktionalität des ER-Editors, auf deren Basis später komplexere Prozessfragmente gebildet werden können. Die Ein- und Ausgabeparameter der Aktionen werden als Situationstypen, die auf den zuvor definierten Produkttypen basieren, modelliert. Zu den typischen Aktionen gehören Erzeugung- und Löschoperationen für die jeweiligen Produkttypen (wie z.B. `createEntity`, `deleteEntity`), aber auch Layout-Operation, die das Produktmodell unverändert lassen. Für den ER-Editor wurde ein Satz von ca. 20 Aktionen definiert.

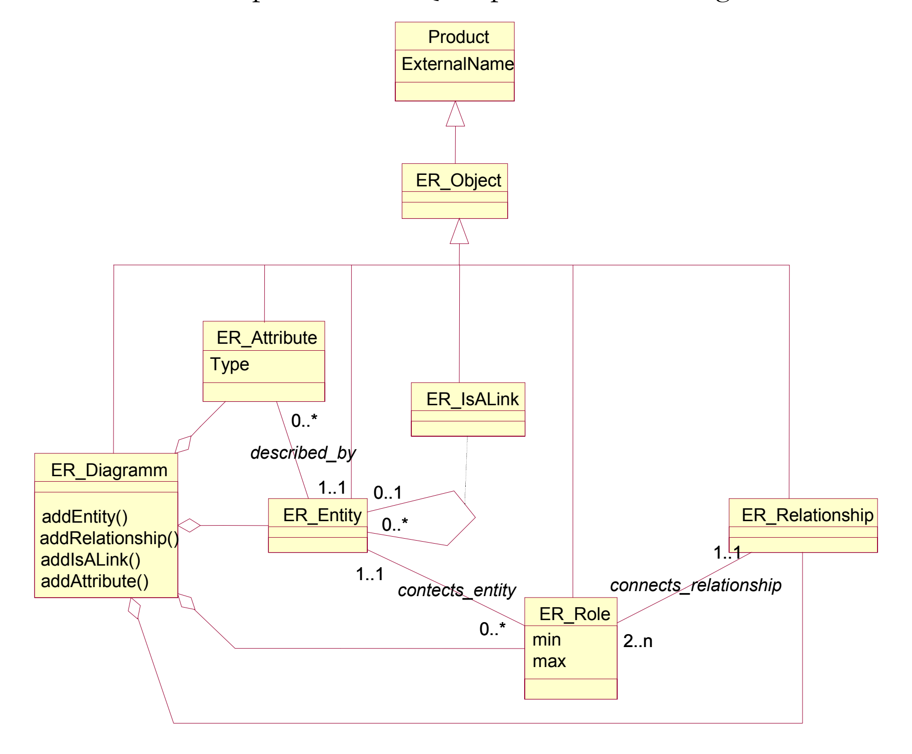
7.3.4.2 Phase 2: Implementierung

Realisierung des Produktmodells

Das Produktmodell wird zunächst in relationales Datenbankschema übertragen. Bei dieser Abbildung wird von den generischen Komponenten des Frameworks lediglich verlangt, dass die Tabellennamen mit den Klassennamen im objektorientierten Produktmodell korrespondieren und dass jede Tabelle über ein Attribut `ID` zu eindeutigen Identifikation und ein Attribut `ExternalName` zur textuellen Darstellung von Instanzen (in generischen Anzeige-Werkzeugen wie z.B. dem Produktmodell-Editor und Abhängigkeitseditor) verfügt. Das relationale Schema wird anschließend durch das Paket `ER_Model` (die ER-Editor-spezifische Ausprägung des Pakets `T_Model`) verkapselt. Dieses Paket implementiert eine objektorientierte Zugriffsschicht auf das Produktmodell. Die Anbindung an ein relationales Daten-

banksystem stützt sich auf Hilfsklassen im Paket ER_DBInterface ab. Dieses Paket spezialisiert die allgemeinen Anfrageklassen im Paket Generic_DBInterface und realisiert die ER-Editor-spezifischen SQL-Updates und -Anfragen[37].

***Abb. 61**: Produktmodell des ER-Editors*



Benutzeroberfläche

Im Paket ER_GUI wird die ER-Editor-spezifische Benutzeroberfläche realisiert. Das Framework stellt für diesen Zweck im Paket Generic_GUI schon einen Großteil der benötigten Funktionalität bereit. Dazu gehört die Erzeugung von Standarddialogen sowie des Hauptfensters, das entsprechend der Werkzeugmodellierung automatisch mit passenden Menüstrukturen, Kommando-Icons, einer Statuszeile und einer Inhaltsregion (Zeichenfläche) angereicht wird. Die Inhaltsregion bietet die Funktionalität eines einfachen grafischen Editors auf Basis eines Prototyp-basierten Entwurfsmusters [GVJH95]. Durch Registrierung einer Menge von vektorbasierten Grundsymbolen (Rechtecke, Kreise, Rauten, Pfeile etc.) erbt die werkzeugspezifische Benutzeroberfläche einen vordefinierten Satz von Basisoperationen (inklusive komplexer Einfügeoperationen, Verschiebe- und Layout-Operationen), ohne dass hierfür zusätzliche Code erforderlich ist. Ein einfaches Werkzeug wie der ER-Editor kommt daher mit einem Minimum spezifischen GUI-Codes aus (ca. 600 Zeilen). Bei komplexeren Werkzeugen, die grafische Symbole, Operationen und Darstellungsarten benötigen, die nicht vom Framework bereit gestellt werden, erhöht sich der Aufwand entsprechend. Beispielsweise umfasst das werkzeugspezifische GUI-Paket des Ziel-Editors der PRIME-CREWS-Umgebung

[37] Da an die Struktur des resultierenden Datenbankschemas außer den beiden oben genannten Attributen keine speziellen Anforderungen gestellt werden, können für die Realisierung und Persistenz des Produktmodells auch automatisierte Generierungstechniken zum Einsatz kommen, wie sie von CASE-Werkzeugen wie etwa Rational Rose oder Together angeboten werden. In diesem Fall würde der Code des Pakets ER_Model automatisch erzeugt und das Paket ER_DBInterface obsolet werden.

[HaPW98; Haum00] knapp 6000 Zeilen, wovon allerdings gut die Hälfte auf generierten Code entfällt.

Aktionen und ObjectTable

Die `Action`-Objekte im Paket `ER_Actions` repräsentieren die eigentliche Benutzerfunktionalität und implementieren die im Werkzeugmodell definierten Werkzeugaktionen. Jedes Aktionsobjekt verfügt über eine `execute`-Methode, die vom `ContextExecutor` bei der Aktivierung des entsprechenden Ausführungskontextes aufgerufen wird und ein Situationsinstanz-Objekt als Parameter erhält und zurück liefert. Der generelle Aufbau der `execute`-Methode ist wie folgt:

- Aus der Eingabe-Situationsinstanz werden (über die entsprechenden Rollenbezeichner) die IDs der Produktmodellinstanzen extrahiert, auf denen die Aktion ausgeführt werden soll.
- In der `ObjectTable` werden mithilfe der IDs die entsprechenden Laufzeit-Objekte des werkzeuginternen Produktmodells und der GUI ermittelt.
- Auf den Produktmodell- und GUI-Objekten werden parallel die erforderlichen Operationen durchgeführt (je nach Funktionalität der zu implementierenden Aktion: Erzeugungs-, Modifikations- oder Löschoperationen).
- Eventuell neu erzeugte Produktmodell- und GUI-Objekte werden der `ObjectTable` unter ihrer eindeutigen ID eingetragen.
- Als Rückgabewert wird eine Situationsinstanz vom dafür im Werkzeugmodell definierten Typ erzeugt und an die IDs der in der Aktion erzeugten oder modifizierten Produktinstanzen gebunden.

Eine Aktion umfasst typischerweise zwischen 20 und 50 Zeilen Code. Für die ca. 20 Aktionen, die der ER-Editor in der aktuellen Version bietet, sind etwa 800 Zeilen Code erforderlich. Das folgende Beispiel zeigt die etwas umfangreichere Aktion `createNewRelationship`.

```
CcxtSitData CeraCreateNewRelationship::execute( CcxtSitData  sd )
{

    PRECONDITION( "correct Sit Type", sd.retSitDesc()->getObjectID() ==
                 oidST_ER_OneOrMoreEntitiesSelected );

    // 1. retrieve list of entity objects to be connected from situation data
    TmapProductDescList lsprodEntities = sd.getProdListData(sSR_ER_EntityList);

    // 2. retrieve GUI and product model objects from object table
    TmapGrObjPtrList lspgoEntities; // GUI objects
    TermEntityList   lspereEntities; // product model objects
    CmapProductDesc  prod; // product ID
    forall ( prod, lsprodEntities) {
       PTmapGrObj pgo = NULL;
       PCermEntity pere = NULL;
       perot->letEntity(pgo, pere, prod ); // perot is the ER object table
       lspgoEntities .append( pgo );
       lspereEntities.append( pere );
    }

    // 3. perform updates in GUI and product model

    // 3a. create relationship in GUI interactively
    TkitString sLabel; TkitInt iX, iY;
    TmapGrObjPtrList lspgoRoleLinks;
    PTmapGrObj pgo = NULL;
```

```
perw->createRelationship(
    pgo, sLabel, lspgoRoleLinks, lspgoEntities, iX, iY);

// 3b. create roles and relationship in product model
TermRoleList lsperlRoles;
PCermRole perl;
PCermEntity pere;
forall (pere, lspereEntities) {
    // create an ermRole object
    perl = new CermRole;
    perm->connectEntityRole(pere, perl);
    lsperlRoles.append(perl);
}

// create product model object for relationship
PCermRelationship perr = perm->insertRelationship(
    lsperlRoles, sLabel, iX, iY);

// 4. insert objects into object table
PTmapGrObj pgoI;
perot->insertRelationship( pgo, perr );
list_item roleitem = lsperlRoles.first();
forall( pgoI, lspgoRoleLinks ) {
    perl = lsperlRoles.contents( roleitem );
    pgo = NULL ;
    perot->addConnection(pgoI, pgo, pgo, perl, perr);
    roleitem = lsperlRoles.succ(roleitem);
}

// 5. create output situation instance
CcxtSitData sdRet;
sdRet.setSitDesc( _perinERED->retCxtMgr()->reloadSituation(
    oidST_ER_OneRelationshipSelected ) );
sdRet.setProdData( perr->getObjectID(),
                   oidPT_ER_Relationship, sSR_ER_Relationship );

return sdRet; }
```

In Schritt 2 und 4 wird die `ObjectTable` benutzt, um zu einer gegebenen Produktinstanz-ID die korrespondierenden Produktmodell- und GUI-Objekte zu ermitteln bzw. um die Korrespondenz zwischen Produktinstanz-ID, Produktmodell- und GUI-Objekten dort abzulegen. Die Klasse `CmapObjectTable` aus dem Framework-Paket `map` bietet hierfür im Prinzip bereits die nötige Funktionalität in Form generischer Methoden. In den meisten Werkzeugen, so auch im ER-Editor, wird diese Klasse jedoch spezialisiert und mit Werkzeug-spezifischen Methoden angereichert, die einen bequemeren Zugriff auf die `ObjectTable`-Funktionalität erlauben (hier z.B. `CersObjectTable::insertRelationship`).

7.3.4.3 Phase 3: Prozessmodellierung

Nach Phase 1 und 2 verfügt der ER-Editor zwar im Prinzip bereits über seine Basisfunktionalität, jedoch ist diese noch nicht in komplexeren Abläufen und für andere Werkzeuge nutzbar.

Dazu müssen zunächst für alle Aktionen im Prozessmodell korrespondierende Ausführungskontexte modelliert werden, wodurch die Funktionalität des ER-Editors auf Prozessmodellierungsebene zugänglich wird. Die Ausführungskontexte können insbesondere in Plankontexte und Entscheidungskontexte anderer Werkzeuge eingebettet werden, so dass der ER-Editor werkzeugübergreifend eingesetzt werden kann.

Mithilfe von Entscheidungskontexten werden die Arbeitsmodi des ER-Editors modelliert. Als Minimalkonfiguration sind ein oder mehrere *Standardkontexte* erforderlich, die je nach vorliegender Situation (z.B. „kein ER-Diagramm geladen“ oder „ER-Diagramm geladen“) die verfügbaren Optionen im Standardarbeitsmodus umfassen. In die Entscheidungskontexte des ER-Editors können durchaus auch Kontexte anderer Werkzeuge, deren Aktivierung aus dem ER-Editor heraus sinnvoll ist, aufgenommen werden. Ein Ausschnitt aus dem integrierten Prozess- und Werkzeugmodell des ER-Editors wurde bereits in Abschnitt 5.6 bei der Beschreibung der Metamodelle vorgestellt.

Komplexere Abläufe, die die Funktionalität des ER-Editors projekt- oder organisationsspezifisch erweitern oder anpassen, werden mithilfe von Plankontexten modelliert. Abb. 62 zeigt ein Beispiel für einen solchen als SLANG-Netz modellierten Plankontext. Dieser Plankontext definiert methodische Anleitung bei der Verfeinerung eines Entitätstypen im Rahmen eines Vorgehensmodell zur Strukturierten Analyse [Your89]. Die Idee des Ablauf besteht darin, möglicherweise von der Verfeinerung betroffene Elemente im korrespondierenden Datenflussdiagramm aufzuspüren und gegebenenfalls anzupassen.

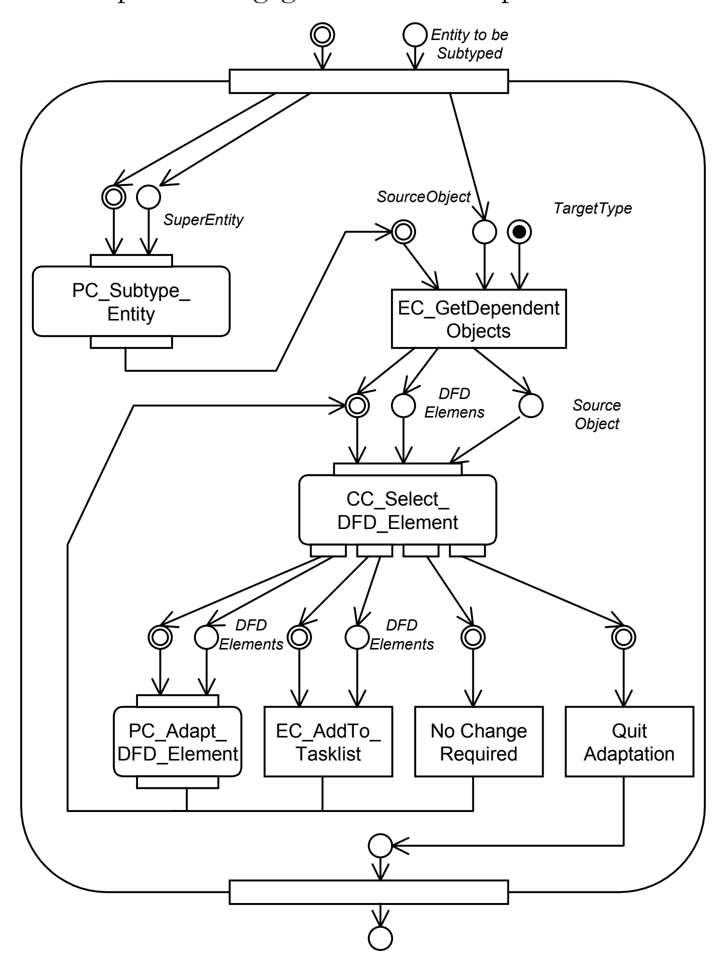


***Abb. 62**: Plankontext für die konsistente Verfeinerung eines Entitätstypen*

Der Ablauf sieht vor, dass nach der Verfeinerung eines Entitätstypen (durch Subtypisierung mithilfe des eingebetteten Plankontexts `PC_SubtypeEntity`) zunächst ermittelt wird, ob im zugehörigen Datenflussdiagramm bestimmte Elemente von dem verfeinerten Entitätstypen abhängen (mithilfe des Ausführungskontexts `EC_GetDependentObjects`, der vom Abhängigkeitseditor angeboten wird). Beispielsweise könnte der Entitätstyp mit einem Datenspeicher oder mit Datenflüssen

korrespondieren. Aus den ermittelten Datenflussdiagramm-Elementen kann der Benutzer ein Element auswählen (Entscheidungskontext CC_Select_DFDElement) und sich zwischen folgenden Alternativen entscheiden:

- Er kann das DFD-Element anpassen (beispielsweise durch Verfeinerung des Datenspeichers oder durch Aufsplitten von Datenflüssen). Dies ist ein Teilablauf, der durch den Plankontext PC_Adapt_DFDElement (hier nicht im Detail dargestellt) angeleitet wird.
- Er kann eine neue Aufgabe zur späteren Erledigung generieren. Diese Funktionalität wird ihm vom Task-Manager der PRO-ART-Umgebung zur Verfügung gestellt (Ausführungskontext EC_AddToTaskList).
- Er kann entscheiden, dass bei dem betreffenden Element keine Anpassung erforderlich ist (NoChangeRequired).
- Er kann den gesamten Anpassungsvorgang beenden (QuitAdaptation).

Das kurze Beispiel demonstriert, wie auf einfache Weise die Funktionalität des ER-Editors, sobald sie erst einmal auf der Ebene der Prozessmodellierung verfügbar ist, in feingranulare, werkzeugübergreifende Abläufe eingebunden werden kann. Insbesondere lassen sich somit projekt- oder organisationsspezifische Prozesse in der Werkzeugumgebung durchsetzen.

7.3.5 Zusammenfassung

Das GARPIT-Framework ist ein wiederverwendbares objektorientiertes Framework, das die Entwicklung prozessintegrierter und anpassbarer Werkzeuge signifikant erleichtert. Es stellt generische Komponenten für die Synchronisation mit der Leitdomäne (StateManager und MessageInterface) und die Interpretation des Umgebungsmodells (ContextManager) zur Verfügung, die das Prozessmodell-konforme Verhalten eines Werkzeugs sicher stellen. Spezifische Werkzeugfunktionalität kann an wohldefinierten Variationspunkten in das Framework eingeklinkt werden. Durch die saubere Kapselung der Anbindung an spezifische GUI-Bibliotheken, Datenbankmanagementsysteme und Kommunikationsmechanismen ist das Framework sehr einfach auf neue Plattformen portierbar. Die Entwicklung eines prozessintegrierten Werkzeugs wurde anhand eines Entity-Relationship-Editors illustriert. Insgesamt wurden bislang 17 Werkzeuge mithilfe des Frameworks erstellt. Dabei betrug der Wiederverwendungsgrad bei dem meisten Werkzeugen mehr als 85 %.

7.4 Integration existierender Werkzeuge

Das GARPIT-Framework ist ursprünglich als ein Ansatz zur *a priori*-Integration von Werkzeugen entwickelt worden. Bisher sind wir in dieser Arbeit stets von einer *Neuentwicklung* der für eine prozessintegrierte Umgebung benötigten Werkzeuge mithilfe des Implementierungsframeworks ausgegangen. Aus wissenschaftlicher Perspektive bestand unser primäres Ziel darin, ein Konzept für die Prozessintegration von Entwurfsumgebungen zu entwickeln und dessen prinzipielle Machbarkeit und Nutzen zu demonstrieren. Der Ansatz einer Framework-basierten Realisierungsplattform stellt dabei ein gewisses Maß sowohl an Entwicklungseffizienz als auch an Robustheit der Implementierung sicher.

In der Praxis stellt die Fokussierung auf die a priori-Integration allerdings eine nur schwer hinnehmbare Einschränkung dar. Viele Entwickler sind nicht bereit, auf die Werkzeuge ihrer gewohnten Arbeitsumgebung zu verzichten, und häufig wird die Verwendung bestimmter kommerzieller Werkzeuge bereits vertraglich durch den Kunden vorgegeben, so dass in der Wahl der Werkzeuge kaum Spielraum besteht [BrMc91; EBLA96]. Darüber hinaus erfordert die Realisierung *spezifischer* Funktionalität bei komplexen Werkzeugen einen erheblichen Aufwand, der auch bei einer stark auf Wiederverwendung ausgerichteten Entwurfsmethodik nicht unterschätzt werden darf.

Ein Ansatz zur Prozessintegration kann also letztendlich nur dann erfolgreich sein, wenn er in der Lage ist, die in einer Organisation vorgefundene Werkzeug-Landschaft einzubeziehen. In diesem Abschnitt beschäftigen wir uns daher mit den Möglichkeiten und Grenzen, existierende Werkzeuge mithilfe geeigneter Verkapselungstechniken („Wrapper“) der Prozessintegration einer PRIME-basierten Umgebung zugänglich zu machen und beschreiben unsere Erfahrungen mit der a posteriori-Integration von drei externen Werkzeugen.

7.4.1 Anforderungen an Werkzeugschnittstellen

Gemäß den in Abschnitt 7.3.1 formulierten Anforderungen an das GARPIT-Framework muss ein prozessintegriertes Werkzeug (1) das Umgebungsmodell interpretieren (Ausführung von angeforderten Entscheidungs- und Ausführungskontexten sowie Erkennung von Kontextaktivierungen) und (2) das Interaktionsprotokoll mit der Leitdomäne zwecks Synchronisation des Prozesszustands realisieren. Ein Werkzeug, das ohne Kenntnis des PRIME-Ansatzes entwickelt wurde, wird diese Anforderungen per se nicht erfüllen können. Daher muss es geeignete Schnittstellen anbieten, damit es dennoch durch einen Wrapper kontrolliert und in seiner Arbeitsweise den Vorgaben aus der Modellierungs- und Leitdomäne unterworfen werden kann.

Inkrementell nutzbare Laufzeit-API erforderlich

Da die Prozessmaschine bzw. der Prozessintegrations-Wrapper mit dem zu integrierenden Werkzeug inkrementell zur Laufzeit interagiert, muss das Werkzeug über offene, programmierbare *Laufzeit*-Schnittstellen (APIs, application programming interfaces) verfügen. Eine Schnittstelle, die lediglich den entkoppelten Zugriff auf die Werkzeugdaten (etwa durch Export in ein proprietäres oder auch standardisiertes Format) ermöglicht, ist nicht ausreichend.

Das Werkzeug muss zum einen sein Objektmodell über eine entsprechende API öffnen. Aus dieser API können das Produktmodell und die darauf operierenden Aktionen abgeleitet werden und innerhalb des Werkzeugmodells nachmodelliert werden. Zum anderen muss der Wrapper den Zustand der Benutzeroberfläche manipulieren können und über relevante Benutzerereignisse informiert werden. Im Einzelnen lässt sich die Funktionalität der benötigten APIs unmittelbar aus dem Umgebungsmodell ableiten. Insgesamt haben wir sechs Kategorien von erforderlichen Laufzeit-APIs identifiziert [Poh*99]:

- **A1: Dienstaufruf**:

  Bei der Aktivierung eines Ausführungskontexts muss der Prozessintegrations-Wrapper die im Umgebungsmodell zugeordnete feingranulare Aktion aktivieren können. Ein Werkzeug sollte daher im Idealfall über seine

Dienstaufruf-API alle Aktionen anbieten, die auch über die interaktive Benutzerschnittstelle genutzt werden können.

- **A2: Ergebnis-Rückmeldung**:

  Über diese Schnittstelle können die Resultate einer Aktionsausführung vom Prozessintegrations-Wrapper erfragt werden. Dies ist erforderlich, da nach der Aktivierung eines Ausführungskontexts die weitere Prozessmodellinterpretation im Allgemeinen von den Ergebnissen der aktivierten Werkzeugaktion abhängt. Wenn die Dienstaufruf-API in der Form *synchroner* (entfernter) Prozedur- oder Methodenaufrufe realisiert ist, fällt diese meist mit der Rückmeldungs-API zusammen. Die Unterscheidung zwischen den beiden APIs macht aber dennoch Sinn, da manche Werkzeuge nur den asynchronen Aufruf von Diensten erlauben oder bis auf einen Fehlercode keine Rückgabewerte liefern.

- **A3: Kommandoerweiterung**:

  Über diese Schnittstelle können zusätzliche Kommandos in die Benutzerschnittstelle eines externen Werkzeugs eingefügt werden und an Rückruf-Funktionen gebunden werden, die im Prozessintegrations-Wrapper die Aktivierung der korrespondierenden Intentionen signalisieren. Die Notwendigkeit dieser Schnittstelle resultiert aus der Eigenschaft des Kontextmodells, dass Entscheidungskontexte extern realisierte Plankontexte und die Kontexte anderer Werkzeuge umfassen können, die somit als Kommandoelemente in der Benutzeroberfläche eines zu integrierenden Werkzeug adäquat abgebildet werden müssen.

- **A4: Produktdarstellung**:

  Über diese Schnittstelle können Produktinstanzen bei der Durchführung von Entscheidungskontexten gemäß der Modellierung im Werkzeugmodell in der Benutzerschnittstelle als hervorgehoben, selektierbar oder deaktiviert dargestellt werden (siehe auch Abschnitt 7.3.2.5).

- **A5: Selektierbarkeit**:

  Über diese Schnittstelle kann bei der Ausführung eines Entscheidungskontexts die Aktivierbarkeit von Produktinstanzen und Kommandos dynamisch eingeschränkt werden, um die Auswahl aktuell nicht erlaubter oder irrelevanter Alternativen durch den Benutzer zu verhindern.

- **A6: Selektionsnotifikation**:

  Über diese Schnittstelle können vom Benutzer ausgelöste Selektionsereignisse (Produktinstanzen oder Kommandos) an den Prozessintegrations-Wrapper gemeldet werden, um im Wrapper den Abgleich mit extern definierten Kontextdefinitionen vornehmen zu können.

Wir haben die Anforderungen an die Funktionalität der Schnittstellen bewusst implementierungsunabhängig formuliert, da man trotz gleicher Schnittstellenfunktionalität bei existierenden Werkzeugen jeweils von unterschiedlichen Schnittstellensignaturen, Bindungsmechanismen, Aufrufprotokollen etc. ausgehen muss.

Bietet ein Werkzeug sämtliche APIs A1 – A6 an, so kann im Prinzip die gleiche Integrationsqualität erreicht werden wie bei einem Werkzeug, das a priori mithilfe des GARPIT-Frameworks entwickelt wurde. Da das Produktmodell, die

Aktionen und die Benutzeroberflächenelemente vollständig offen liegen, kann das Werkzeug zunächst im Werkzeugmodell nachmodelliert und darauf aufbauend in Prozessfragmente eingebunden werden.

Eingeschränkte Formen der Prozessintegration

Der so erreichbare Grad der Prozessintegration ist als idealtypisch anzusehen. Beim Fehlen ein oder mehrerer APIs lassen sich jedoch noch abgeschwächte Formen der Prozessintegration realisieren. Beispielsweise kann auch ohne die GUI-bezogenen APIs A3 – A6 immer noch eine Integration erreicht werden, bei der die Prozessmaschine feingranular Ausführungskontexte im zu integrierenden Werkzeug ansteuern kann. Fehlt nur die API A3 (Kommandoerweiterung), ist sogar eine prozesssensitive Anpassung der Interaktionsmöglichkeiten, d.h. die Prozessmaschinen-gesteuerte Durchführung von Entscheidungskontexten in dem Werkzeug möglich. Allerdings können die Entscheidungskontexte dann nur die originäre Funktionalitäten des zu integrierenden Werkzeugs als Alternativen enthalten, da keine zusätzlichen Kommandoelemente für extern definierte Plankontexte oder Kontexte anderer Werkzeuge in die Benutzeroberfläche eingefügt werden können.

7.4.2 Wrapper-Architektur

Der Prozessintegrations-Wrapper vermittelt die Interaktion zwischen der Prozessmaschine und dem zu integrierenden Werkzeug. Ihm fallen dabei zwei wesentliche Aufgaben zu. Er interpretiert erstens die für das Werkzeug relevanten Anteile des Umgebungsmodells, um Kontextaktivierungen (durch die Prozessmaschine oder den Benutzer selbst) auf entsprechende Aktionsaufrufe oder Einschränkungen der Interaktionsmöglichkeiten umzusetzen und Benutzerereignisse mit existierenden Kontextdefinitionen abzugleichen und gegebenenfalls den Aufruf eines aktivierten Kontexts zu initiieren. Zweitens wickelt der Wrapper die Kommunikation mit der Prozessmaschine gemäß dem in Abschnitt 7.2 beschriebenen Interaktionsprotokoll ab.

Diese beiden Aufgaben entsprechen exakt den Funktionen, die im GARPIT-Framework von den Teilsystemen `ContextManager` bzw `StateManager/MessageInterface` wahrgenommen werden. Daher liegt es nahe, das GARPIT-Framework als Basis eines generischen Wrapper-Frameworks weiterzuverwenden. Als äußerst hilfreich erweist sich hier, dass in der GARPIT-Architektur der generische Interpreterkern, der von diesen Teilsystemen gebildet wird, sauber von den werkzeugspezifischen Architekturbausteinen separiert wurde. Abb. 63 zeigt die resultierende generische Architektur des Prozessintegrations-Wrappers.

Anbindung des Fremdwerkzeugs über Action- und UI-Adapter

Man erkennt, dass wesentliche Teile des GARPIT-Frameworks übernommen werden konnten. Neben dem `StateManager/MessageInterface` und dem `ContextManager` sind dies die Teilsysteme `ActionTable`, `IntentionTable` und `ObjectTable`, die dem `ContextManager` eine abstrakte Schnittstelle zur Aktivierung von Aktionen und zur Interaktion mit der Benutzeroberfläche zur Verfügung stellen. Als wesentliche neue Komponenten sind die Teilsysteme `ActionAdapter` und `UIAdapter` hinzugekommen, die die korrespondierenden Teilsysteme `T_Actions` und `T_GUI` der originalen GARPIT-Architektur ersetzen.

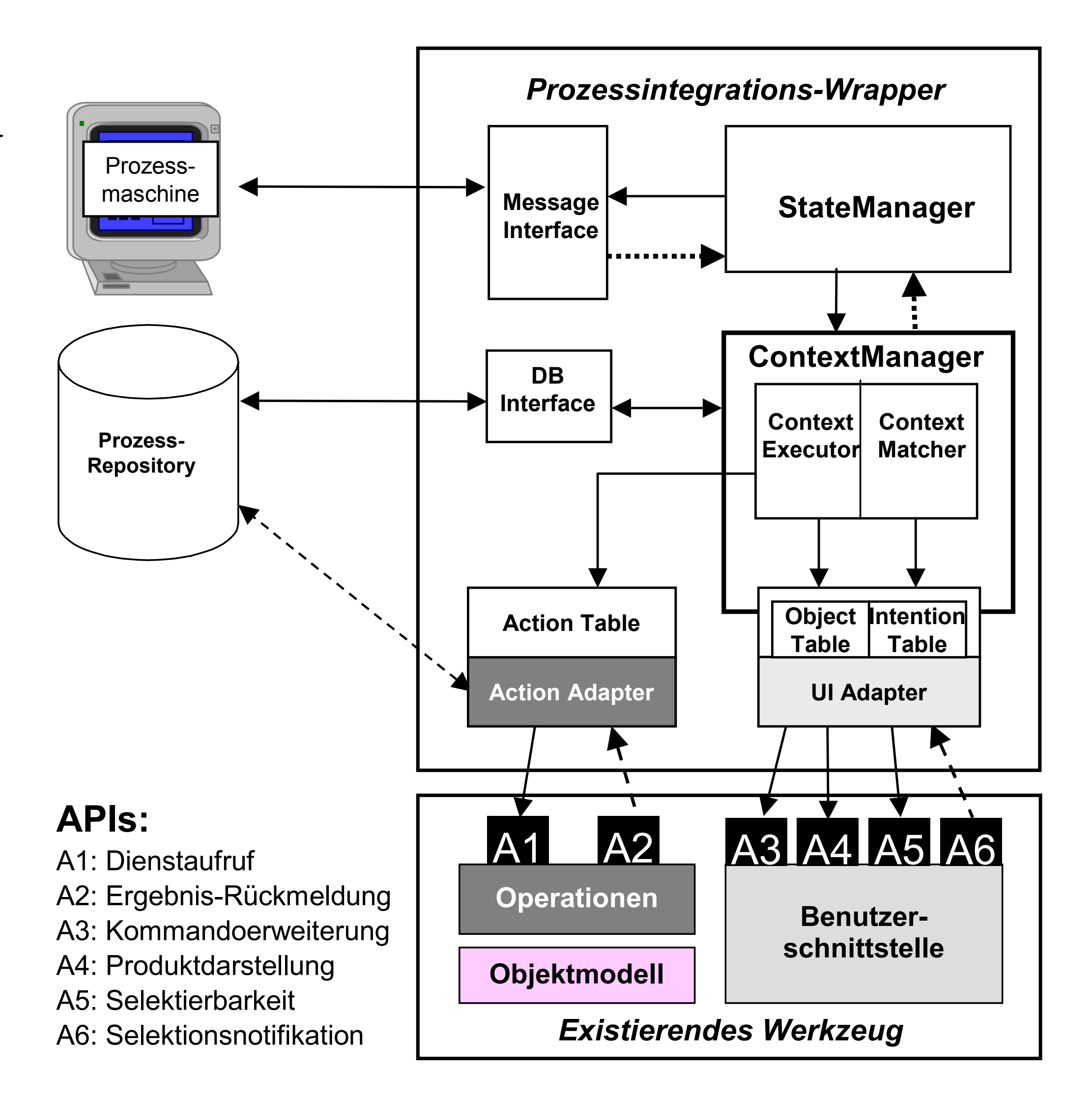


***Abb. 63**:*
Generische Architektur des Prozessintegrations-Wrappers

Das Teilsystem `ActionAdapter` bildet die Aktionen, die im Werkzeugmodell definiert wurden, auf die Dienstschnittstelle des externen Werkzeugs ab und nimmt Rückmeldungen über die Ausführungsresultate entgegen. Dazu nutzt es die oben definierten Schnittstellen A1 und A2. Der `ActionAdapter` ist als eine Sammlung von Klassen organisiert, die von der Klasse `Action` (siehe auch Abschnitt 7.3.2.2) abgeleitet wurden und daher direkt in die `ActionTable` eingetragen werden können. Anders als die `Action`-Klassen im originalen GARPIT-Framework fungieren die `ActionAdapter`-Klassen jedoch nur als Stellvertreter (*Proxies*), die den eigentlichen Aktionsaufruf an das zu integrierende Werkzeug delegieren. Dabei nutzen sie die jeweils spezifischen Aufrufmechanismen (CORBA, COM, SOAP o.ä.). Eine wichtige Aufgabe besteht in der Konvertierung zwischen den als Situationsinstanzen definierten Ein- und Ausgabedaten einerseits und den von der Werkzeug-API vorgegebenen Datenformaten andererseits.

Das Teilsystem `UIAdapter` kommuniziert Änderungen zwischen der `ObjectTable` und der `IntentionTable` und der Benutzeroberfläche des externen Werkzeugs. Beispielsweise werden bei der Ausführung eines Entscheidungskontextes über die APIs A3 und A5 die Kommandoelemente des Werkzeugs auf die Intentionen der aktuell auswählbaren Alternativen angepasst. Mithilfe von API A4 werden die Produkte der aktuellen Situationsinstanz in der Benutzeroberfläche hervorgehoben. Über die API A6 werden vom Benutzer ausgelöste Selektionsereig-

nisse an den `UIAdapter` zurück gemeldet und von dort an die `ObjectTable`/`IntentionTable` weitergereicht, woraufhin eine Kontextauswertung angestoßen wird. Wie beim `ActionAdapter` besteht die Hauptaufgabe des `UIAdapters` somit in der Kapselung der werkzeugspezifischen APIs und Aufrufmechanismen.

7.4.3 Validierung

Erste Anwendungserfahrungen hinsichtlich des beschriebenen a posteriori-Integrationskonzepts und der generischen Wrapper-Architektur konnten wir im Rahmen des Sonderforschungsbereichs IMPROVE sammeln. Ziel dieses Projekt ist die umfassende Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik. Das für diese Arbeit relevante SFB-Teilprojekt „Erfahrungsbasierte Prozessunterstützung kooperativer Entwicklungsprozesse" beschäftigt sich mit der feingranularen Entwickleranleitung in einer verfahrenstechnischen Modellierungsumgebung, die aus einer Vielzahl kommerzieller Entwurfs-, Simulations- und Dokumentationswerkzeugen besteht [WeBa99]. Wir beschreiben in Kapitel 8 noch ein detailliertes Anwendungsbeispiel aus dem IMPROVE-Umfeld und beschränken uns daher hier auf die in diesem Projekt mit insgesamt drei Werkzeugen durchgeführten Integrationsexperimente:

- **Aspen Plus**: Aspen Plus ist das marktführende Programmpaket für die statische Simulation chemischer Prozessmodelle.
- **Microsoft Excel**: MS Excel wird häufig in der frühen Phase der Prozessentwicklung für die Grobberechnung von Massenbilanzen sowie für Kostenkalkulationen verwendet.
- **Visio**: Visio ist ein weit verbreitetes Werkzeug für technische Zeichnungen, das in der Verfahrenstechnik hauptsächlich zur Erstellung von Fließbilddiagrammen, dem Hauptdarstellungsmittel für chemische Prozesse bzw. Anlagen, eingesetzt wird.

Aspen Plus: lose Integration, Anleitung in separater Benutzerschnittstelle

Der geringste Integrationsgrad wurde bei Aspen Plus erreicht. Aspen Plus ist ein weitgehend geschlossenes, monolithisches System, das nicht über geeignete Programmierschnittstellen zur Prozessintegration verfügt. Aspen Plus legt zwar einen Teil seines Objektmodells über COM-basierte APIs offen, jedoch erlauben diese nur lesenden Zugriff, so dass beispielsweise die Prozessmaschinen-gesteuerte Erzeugung eines neuen Simulationsmodells nicht möglich ist. Die unmittelbare Einflussnahme der Prozessmaschine auf Aspen Plus beschränkt sich daher auf das Starten des Werkzeugs (ggf. mit einer über die Kommandozeile übergegebenen Simulationsdatei). Arbeitsabläufe, die Aspen Plus einbeziehen, werden stattdessen über das in Abschnitt 7.1.2.5 beschriebene generische Anleitungswerkzeug unterstützt. In diesem Werkzeug wird der Benutzer über die aktuell anwendbaren Arbeitsschritte informiert und kann die Ergebnisse von Aktionen (Ausführungskontexte) oder seine Auswahl unter mehreren möglichen Alternativen (Entscheidungskontexte) an die Prozessmaschine zurückmelden. Der automatisierte Aufruf von Aktionen oder die unmittelbare Anpassung der Interaktionsmöglichkeiten in Aspen Plus ist so jedoch nicht möglich.

Microsoft Excel: Prozessintegration auf der Ebene von Ausführungskontexten

Bei Microsoft Excel haben wir die Prozessmaschinen-gesteuerte Aktivierung von feingranularen Aktionen betrachtet (also Prozessintegration auf der Ebene von Ausführungskontexten über die APIs A1 und A2). Diese Beschränkung ergab sich primär aus einer Analyse der in Excel zu unterstützenden Arbeitsabläufe. In

den von uns betrachteten Anwendungsszenarien wird aus den Daten eines Fließbildmodells eine Excel-Tabelle für die Berechung von Massenbilanzen und Kosten erstellt. Dieser Vorgang wird mithilfe eines Plankontexts automatisiert, der die als Ausführungskontexte modellierten Excel-Funktionalitäten (z.B. Einfügen eines Wertes oder einer Formel in eine Tabellenzelle) ansteuert. Nach der automatisierten Erstellung der Tabellen finden in Excel selbst jedoch keine prozessrelevanten Abläufe mehr statt, so dass eine Anpassung der Interaktionsmöglichkeiten in diesem Werkzeug nicht erforderlich war. Wir wollen jedoch betonen, dass Microsoft Excel (ebenso wie die anderen Programme des MS Office-Pakets) über seine COM-Schnittstelle alle benötigten APIs offen legt, so dass im Prinzip eine ähnlich vollständige Prozessintegration wie beim nachfolgend beschriebenen Werkzeug Visio möglich gewesen wäre.

Visio: vollständige Prozessintegration

Unser Hauptinteresse bei den Prozessintegrationsexperimenten galt Visio, das die technische Grundlage für den IMPROVE-Fließbildeditor bildet[38]. Dieses Werkzeug nimmt innerhalb des IMPROVE-Werkzeugverbunds eine zentrale Stellung ein, da der Umgang mit fließbildartigen Abstraktionen einer chemischen Anlage im Mittelpunkt einer Vielzahl verfahrenstechnischer Entwicklungsprozesse steht. Eine Analyse dieser Prozesse ergab, dass abhängig vom Arbeitsablauf für den Benutzer nur ein spezifischer Teil der Visio-Funktionalität relevant ist und außerdem eine Reihe von Prozessfragmenten aus dem Fließbildwerkzeug heraus angestoßen werden. Aus diesem Grund war für uns neben der Kontrolle von elementaren Visio-Aktionen auch die prozesssensitive Anpassung der Interaktionsmöglichkeiten, d.h. die Durchsetzung von Entscheidungskontexten, und die Aktivierbarkeit von Plankontexten aus Visio heraus essentiell.

Auf der Modellierungsebene wurden die zu kontrollierenden Aktionen, Produkte und Kommandoelemente des Visio-basierten Fließbild-Editors mit den Konzepten des Werkzeugmetamodells beschrieben. Aufbauend auf diesen Definitionen konnten geeignete Ausführungs-, Entscheidungs- und Plankontexte definiert werden.

Der generische Prozessintegrations-Wrapper wurde um spezialisierte Adapterklassen in den Paketen `ActionAdapter` und `UIAdapter` angereichert, die sich auf die von Visio bereitgestellten COM-Schnittstellen abstützen. Bei der ansonsten unkomplizierten Implementierung stießen wir auf eine Reihe von Detailproblemen, die wir hier kurz beschreiben, um die typischen „Fußangeln" einer a posteriori-Integration zu illustrieren:

- Jedes Visio-Kommando kann potenziell in 14 verschiedenen Menüs bzw. Icon-Leisten auftauchen, die folglich alle vom Prozessintegrations-Wrapper kontrolliert werden müssen. Dadurch wird der Adapter zwischen der entsprechenden Visio-API und der `IntentionTable` aufgebläht und schwerer wartbar.
- In Visio können Zeichenelemente mittels „Drag & Drop" von einer Vorlage auf die Zeichenfläche gezogen und dort eingefügt werden. Dieser Vorgang entspricht in der PRIME-Terminologie der Aktivierung eines Kontexts (Auswahl des Zeichenelements mit impliziter Intention „Create" und „leerer" Situation) und der anschließenden Durchführung eines

[38] Die Erweiterung von Visio um verfahrenstechnische Funktionalitäten wird in Kapitel 8 detailliert beschrieben.

Ausführungskontexts (Einfügen des Objekts in die Zeichenfläche). In Visio ist dieser Vorgang aus Sicht des Wrappers atomar, d.h. der Wrapper wird erst nach dem Einfügen des Zeichenelements in das aktuelle Diagramm durch ein entsprechendes Ereignis informiert. Dies konfligiert jedoch mit dem GARPIT-Modell der Ereignisabarbeitung, bei der nach der Intentionsaktivierung erst der `ContextMatcher` aktiviert wird und von diesem dann die eigentliche Kontextausführung angestoßen wird. Dies soll von vorneherein eine Abweichung von der intendierten Prozessanleitung verhindern, die sich dann ergäbe, wenn die Einfüge-Aktion nicht zu den erlaubten Alternativen im aktuellen Entscheidungskontext gehörte und eigentlich gar nicht hätte durchgeführt werden dürfen. In der Visio-Integration haben wir dieses Problem dadurch umgangen, dass nach dem Auslösen des Ereignisses „Element eingefügt" die durchgeführte Aktion mit Hilfe des ContextManagers nachträglich auf ihre Zulässigkeit geprüft und gegebenenfalls unter Ausnutzung der von Visio angebotenen Undo-Funktion rückgängig gemacht wird.

- Wir hatten ursprünglich geplant, Visio und den Prozessintegrations-Wrapper als zwei getrennte Betriebssystemprozesse zu realisieren, wobei Visio als *COM Automation* Server fungieren und über seine *Automation* Schnittstelle durch den Prozessintegrations-Wrapper kontrolliert werden sollte. In diesem Modus bietet Visio einem externen Klienten jedoch aus einem für uns nicht ersichtlichen Grund nicht die Möglichkeit, sich für den Erhalt von Menü-Ereignissen zu registrieren. Diese Funktion, die essentiell für die Erkennung von Kontextaktivierungen ist (API A6), ist nur für eine so genannte „in-process extension" zugreifbar. Aus diesem Grund waren wir gezwungen, den Prozessintegrations-Wrapper in die Form einer DLL (dynamic link library) zu transformieren, die beim Starten durch einen dynamischen Lademechanismus zu Visio hinzugebunden wird.
- Eine Konsequenz der Verschmelzung von Visio und des Prozessintegrations-Wrappers zu einem Betriebssystemprozess war, dass wir zwei konkurrente Ereignisschleifen erhielten: eine in Visio und eine im Prozessintegrationswrapper (für den Empfang von Nachrichten aus der Leitdomäne). Da sich diese beiden Ereignisschleifen anfangs gegenseitig blockierten, mussten wir die Ereignisschleife des Prozessintegrations-Wrappers leicht modifizieren.

Trotz dieser Probleme war die Realisierung des Prozessintegrations-Wrappers und die Integration mit den generischen Framework-Klassen insgesamt relativ einfach und „kanonisch". Die geschilderten technischen Schwierigkeiten fallen in die gleiche Kategorie von Problemen, die auch andere Autoren [GaAO95; MaBF99; Ble*99a] bei der Integration unabhängig voneinander entwickelter Frameworks gemacht haben. Wir glauben daher, dass die Implementierungsprobleme weniger Ausdruck einer grundsätzlichen konzeptionellen Schwäche unseres Modellierungs- und Integrationsansatzes sind, sondern unvermeidlich sind bei der Integration größerer Software-Einheiten.

7.4.4 Zusammenfassung

In diesem Abschnitt haben wir gezeigt, dass der PRIME-Ansatz keineswegs auf a priori-Integrationsszenarien beschränkt ist, sondern sehr wohl auch bei existieren-

den Werkzeugen angewandt werden kann und dort zu einer deutlichen Verbesserung der Integrations- und Prozessunterstützungsqualität beitragen kann.

Wir haben insgesamt sechs API-Kategorien identifiziert, die ein Werkzeug anbieten muss, damit es über einen Wrapper vollständig prozessintegriert werden kann. Für den Prozessintegrations-Wrapper konnten große Teile des GARPIT-Frameworks wiederverwendet werden. Signifikant erleichtert wurde dies durch die saubere Trennung zwischen dem generischen Interpreterkern und den Anschlussstellen für die werkzeugspezifischen Funktionalitäten in der GARPIT-Architektur. Die Anbindung an ein externes Werkzeug erfolgt über Adapterklassen, die zwischen den Variationspunkten des generischen Prozessintegrations-Wrappers und den spezifischen Werkzeug-APIs vermitteln.

Der bei den Integrationsexperimenten erreichte Grad der Prozessintegration differiert zwischen den betrachteten Werkzeugen Aspen Plus, Microsoft Excel und Visio. Dies haben wir zum einen mit der unterschiedlichen Offenheit der Werkzeuge begründet und zum anderen mit den spezifischen Rollen, die die Werkzeuge in den zu unterstützenden Prozessen spielen. Während sich für Microsoft Excel eine Prozessintegration auf der Ebene von Ausführungskontexten als ausreichend erwies, wurde bei Visio der Zusatzaufwand für eine vollständige Prozessintegration durch die zentrale Stellung des Werkzeugs im IMPROVE-Werkzeugverbund gerechtfertigt.

Die hier beschriebenen Experimente sollten die grundsätzliche Machbarkeit der Prozessintegration existierender Werkzeuge demonstrieren und sind lediglich als erster Schritt auf dem Weg zu einem umfassenden a posteriori-Integrationskonzept im PRIME-Kontext zu verstehen. Es ergeben sich eine Reihe interessanter Anknüpfungspunkte für weiter gehende Forschungsarbeiten. Beispielsweise erfolgt gegenwärtig die Modellierung eines externen Werkzeug im Werkzeugmodell und die Implementierung der entsprechenden Adapter *manuell*. Hier ist die Möglichkeit zu untersuchen, ob aus einer gegebenen, formalen Schnittstellenbeschreibung des zu integrierenden Werkzeugs (z.B. in Form einer IDL-Spezifikation oder einer COM-Type Library) zumindest semiautomatisch entsprechende PRIME-Werkzeugmodelle und Adapterklassen generiert werden können. Wünschenswert wäre weiterhin ein strukturiertes Rahmenwerk, das unterschiedliche Kategorien der Prozessintegrierbarkeit genauer klassifiziert und dem Umgebungsintegrator eine Checkliste für die Bewertung des Integrationspotenzials eines externen Werkzeugs an die Hand gibt. Weiterhin sind die Konsequenzen einer unvollständigen Prozessintegration auf Aspekte wie die Synchronisation zwischen Leit- und Durchführungsdomäne sowie die Protokollierung von Nachvollziehbarkeitsinformationen genauer zu beleuchten. Ein Teil dieser Fragestellungen wird zur Zeit in der zweiten Förderperiode des Sonderforschungsbereichs IMPROVE bearbeitet.

7.5 GARPEM: die generische Prozessmaschinenarchitektur

In diesem Abschnitt beschreiben wir das generische Prozessmaschinen-Framework GARPEM, das den Kapitel 6 vorgestellten Ansatz zur interoperablen Verwendung von Prozesssprachen realisiert. Wir geben zunächst einen Überblick über die Grobstruktur des Frameworks (Abschnitt 7.5.1) und gehen genauer auf die

sprachlichen und technischen Klassen ein (Abschnitt 7.5.2). In Abschnitt 7.5.3 beleuchten wir das Kontrollmodell, das die Zusammenarbeit zwischen dem administrativen Framework-Teil und den spezifischen Sprachinterpretern regelt. In Abschnitt 7.5.4 zeigen wir Parallelen zu verwandten Ansätzen auf.

7.5.1 Grobstruktur des GARPEM-Frameworks

Abb. 64 zeigt die Grobstruktur des GARPEM-Frameworks, das in drei Schichten unterteilt ist. Die *Schnittstellenschicht* ist für die Anbindung der Prozessmaschine an die Durchführungs- und Modellierungsdomäne zuständig.

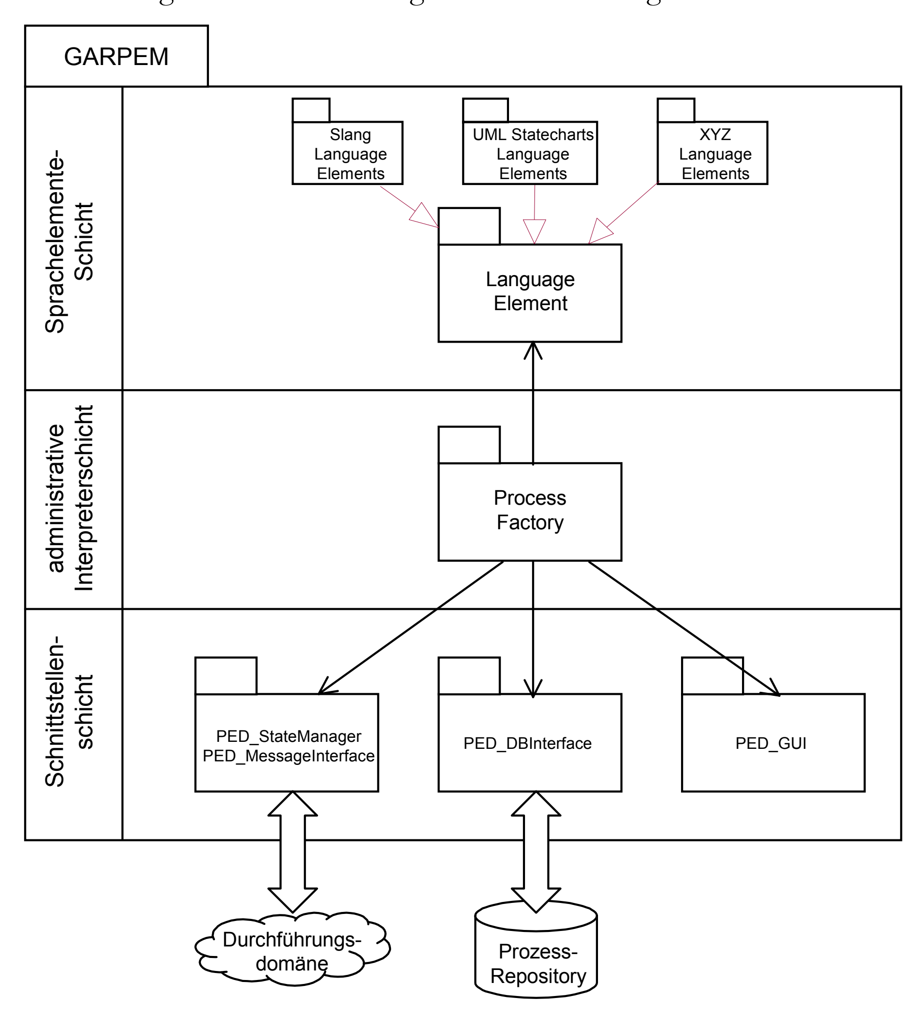


***Abb. 64**: Grobarchitektur des GARPEM-Frameworks*

Die Pakete PED_StateManager[39] und PED_MessageInterface realisieren die globale Zustandsverwaltung der Prozessmaschine und die Nachrichtenschnittstelle zur Durchführungsdomäne gemäß dem in Abschnitt 7.2 vorgestellten Interaktionsprotokoll. Das Paket PED_DBInterface realisiert den Zugriff auf die im Prozess-Repository abgelegten Prozessdefinitionen. Außerdem ist in der Schnittstellenschicht das Paket PED_GUI angesiedelt, das einige rudimentäre Benutzeroberflächenfunktionen für den Start und die Benutzerkontrolle der Prozessmaschine bereitstellt. Die Pakete der Schnittstellenschicht basieren auf den gleichen Basisklassen wie ihre Pendants im generischen Werkzeugframework GARPIT (vgl.

[39] Das Präfix PED steht für **P**rocess **E**nactment **D**omain (Leitdomäne).

Abschnitt 7.3.2); wir verzichten daher auf eine genauere Beschreibung dieser Teilsysteme.

Die eigentliche Prozessfragmentinterpretation findet der *administrativen Interpreter-Schicht* und in der *Sprachelemente-Schicht* statt. Die Sprachelemente-Schicht stellt eine Laufzeitdarstellung der aktuell ausgeführten Prozessfragmente bereit, indem jedes in einem Prozessfragment auftauchende Sprachelement durch ein entsprechendes Objekt repräsentiert wird. Die Methoden dieser Objekte realisieren die operationale Semantik der jeweiligen Sprachelemente. Mithilfe dieser Methoden interpretieren sich die Sprachelemente sozusagen selbst.

Die administrative Interpreter-Schicht ist für die Ausführungskoordination der Sprachelemente zuständig. Dies umfasst das Laden, Instanziieren und Entladen der Objekte der Sprachelemente-Schicht, die Vermittlung der Interoperation zwischen unterschiedlichen Sprachelementen bei geschachtelten Prozessfragmenten und die Anbindung an die Funktionalitäten der Schnittstellenschicht (Kommunikation, Datenbank, Benutzeroberfläche).

In der nachfolgenden Darstellung konzentrieren wir uns auf die Struktur der Sprachelemente-Schicht und der administrativen Interpreter-Schicht. Abschnitt 7.5.2 beschreibt die Klassen der beiden Schichten, während Abschnitt 7.5.3 auf das Kontrollmodell eingeht, das dem Zusammenspiel zwischen Sprachelemente-Schicht und administrativer Interpreter-Schicht zugrunde liegt.

7.5.2 Beschreibung der wiederverwendbaren Klassen

Im Folgenden erläutern wir die wiederverwendbaren Elemente des Interpreter-Rahmenwerkes. In Abschnitt 7.5.2.1 werden zunächst die *sprachlichen* Klassen der Sprachelemente-Schicht vorgestellt, die die Struktur der Prozessmodelle der Modellierungsdomäne abbilden. In Abschnitt 7.5.2.2 werden dann die *technischen* Klassen der administrativen Interpreterschicht beschrieben, die für Ausführung eines Prozessmodells in der Leitdomäne erforderlich sind. Anschließend wird die Vorgehensweise bei der Integration eines Interpreters für eine neue Sprache kurz erläutert (Abschnitt 7.5.2.3).

7.5.2.1 Sprachliche Klassen

Abbildung der PSM2-Konzepte aus Abschnitt 6.4 auf Klassenstruktur der Sprachelemente-Schicht

Im GARPEM-Interpreterrahmenwerk bilden die grundlegenden Sprachkonzepte, die in Abschnitt 6.4.1 innerhalb des Prozesssprachen-Metametamodells (PSM2-Modell) formalisiert wurden, die Basis für die Wiederverwendung. Dort wurden diejenigen Prozesssprachen-Konzepte identifiziert, die für die komponentenbasierte Integration einer Sprache erforderlich sind (im Wesentlichen `Aktivität`, `Schnittstelle`, `Datenbehälter`). Die PSM2-Konzepte und deren Assoziationen werden direkt auf die Klassenstruktur der Sprachelemente-Schicht abgebildet (siehe Abb. 65, vgl. auch Abb. 29 auf Seite 154). Auf diese Weise ist gewährleistet, dass die Sprachen, die sich für eine Integration grundsätzlich eignen, auch durch das Interpreterrahmenwerk unterstützt werden. Zusätzlich wird das Konzept eines Wächters und eines Typs vom Interpreterrahmenwerk angeboten.

Die grundsätzliche Klassenstruktur wurde bereits im Zusammenhang mit den Erläuterungen des PSM2-Modells in Abschnitt 6.4.1 begründet. Anstelle einer detaillierten Beschreibung der Schnittstellen werden im Folgenden die Verantwort-

lichkeiten der einzelnen Klassen dargelegt. Ausgehend von der Basisklasse `LanguageElement` werden alle sprachlichen Konzepte sowohl des Interpreterrahmenwerkes selbst, als auch zusätzliche sprachliche Konzepte einer spezifischen Prozesssprache spezialisiert.

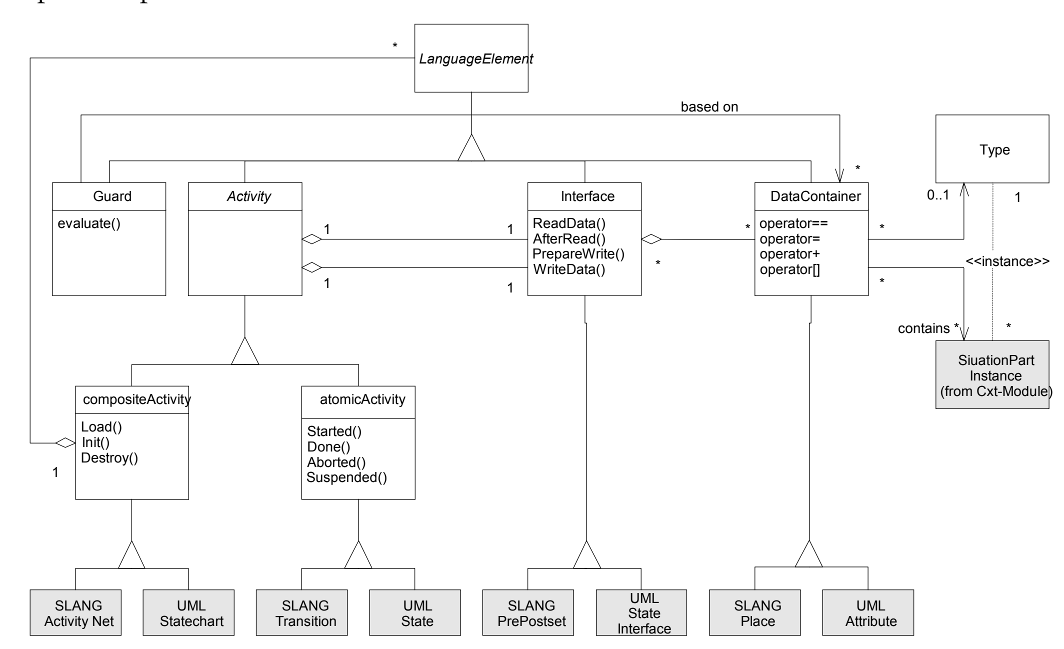


Abb. 65: Sprachliche Klassen des Sprachelemente-Schicht

Das Wissen über die einzelnen Sprachelemente wird von der Klasse `compositeActivity` gehalten. Eine zusammengesetzte Aktivität, d.h. ein Prozessfragment, besteht aus Sprachelementen, die das Verhalten des Fragmentes spezifizieren, und ist für deren Instanziierung und Initialisierung verantwortlich. Die Schnittstelle der Klasse `compositeActivity` bietet daher Funktionalität zum Laden, Initialisieren und Entladen eines Prozessfragmentes und darüber hinaus Funktionalität zum Navigieren innerhalb der Sprachelemente. Da die Struktur eines Fragments bereits durch das PSM2-Modell weitestgehend vorgegeben ist – ein Fragment besteht aus Aktivitäten, Schnittstellen, Datenbehältern und Datentypen – , konnte bei der Implementierung des Interpreterrahmenwerkes die Ladefunktion in weiten Teilen sprachneutral realisiert werden, so dass sie für alle Prozesssprachen wiederverwendbar ist. Eine Anpassung ist lediglich für Prozesssprachen-spezifische Modellelemente erforderlich, die nicht direkt Teil des Interpreterrahmenwerkes sind.

Die Produktdaten innerhalb eines Fragments werden von der Klasse `DataContainer` gehalten, die einen Typ haben können. Die Klasse `Type` ist verantwortlich für die Verwaltung der Typinformationen und stellt Metainformationen über ihre Instanzen bereit. Dies umfasst Anfragefunktionalität nach Generalisierungs- und Spezialisierungsbeziehung zwischen Typen sowie nach Typattributen. Die eigentlichen Produktinstanzen werden in Instanzen der Klasse `CcxtSituationPartInstance` verkapselt, also mithilfe der gleichen Klasse, mit der auch in der Durchführungsdomäne Produktinstanzen an Situationsteile gebunden werden (vgl. Abschnitt 7.3.2.5). So können auf einfache Weise Situationsdaten, die in Werkzeugen gebunden wurden, mit den Datenbehälterkonzepten der jeweiligen Prozesssprachen in Beziehung gesetzt werden. Die Klasse `DataContainer` bietet generell Funktionalität, um Operationen auf Daten auszuführen und ggf. mit Werten zu initialisieren. Typische Funktionen sind hier Zuweisungs- und Vergleichsoperationen sowie Konstruktorfunktionen für Felder und Listen. Diese Basisfunktionen sind für viele Prozesssprachen identisch und werden vom Interpreterrahmenwerk

weitgehend sprachneutral bereitgestellt, so dass hier der Wiederverwendungsgrad hoch ist.

Aktivitäten (sowohl zusammengesetzte als auch atomare Aktivitäten) haben eine Schnittstelle, die durch die Klasse `Interface` repräsentiert wird. Ein `Interface` ist allgemein für die Produktdaten-Ein- und Ausgabe einer Aktivität verantwortlich und im Speziellen für die Transformation von Daten in das sprachspezifische Format des Fragments zuständig. Wie bereits in Abschnitt 6.3.1 erläutert, entspricht ein Interface einer Aktivität der Stellvertreterschnittstelle einer AK-, EK- oder PK-Kontextkomponente. Dabei ist die Ausführung einer Aktivität an die Ausführung eines Kontextes in der PRIME-Umgebung gebunden. Bei der Ausführung wird eine Transformation zwischen der Prozesssprachen-spezifischen Darstellung der aktuellen Situationsdaten in die PRIME-spezifische Darstellung vollzogen. Dazu werden die aktuellen Schnittstellenparameter in das sprachneutrale Format der PRIME-Umgebung transformiert, versandt und auf der Empfängerseite wieder entschlüsselt. Auf diese Weise ist die Interoperabilität verschiedensprachlicher Fragmente gewährleistet, da das Interface die Abbildung der Situationsdaten von einer sprachneutralen in eine sprachspezifische Darstellung und umgekehrt definiert.

Transformation der Situationsdaten

Über diese Transformation hinaus bietet eine Schnittstelle Variationspunkte an, die festlegen, wie die Daten der Schnittstellenparameter vor und nach Schreib- und Leseoperationen modifiziert werden. Vor der Ausführung einer Aktivität werden die aktuellen Daten der Eingangsschnittstelle gelesen und anschließend die Ergebnisse der Ausgangsschnittstelle zugewiesen. Dies führt zu einer sprachabhängigen Modifikation der Inhalte der Datenbehälter beim Zugriff. So werden z.B. im Falle von SLANG beim Schalten einer Transition die Marken im Vorbereich konsumiert, so dass die aktuellen Daten der Eingangsschnittstelle nach der Ausführung nicht mehr referenziert werden können. Andere Sprachen modifizieren die Daten beim Zugriff nicht oder erfordern u.U. referenzielle Semantiken. Durch Anpassung der Variationspunkte kann jeweils die erforderliche Sprachsemantik umgesetzt werden.

7.5.2.2 Technische Klassen

Die zuvor erläuterten sprachlichen Klassen repräsentieren Konzepte der Modellierungsdomäne. Für die Ausführung und sprachunabhängige Erzeugung eines Prozessfragmentes in der Leitdomäne sind darüber hinaus weitere *technische* Klassen der administrativen Interpreterschicht erforderlich. Diese werden von den Klassen `Process` und `Factory` sowie deren Spezialisierungen abgedeckt (siehe Abb. 66).

Factory: Framework-gesteuerte Erzeugung spezifischer Sprachelement-Objekte

Die Klasse `Factory` ist eine Fabrik für die einzelnen Sprachelemente des Interpreterrahmenwerkes und folgt dem *Fabrik*-Entwurfsmuster von [GHJV95]. Die von der Fabrik bereitgestellte Schnittstelle ermöglicht das Erzeugen spezialisierter Sprachelemente, ohne auf den konkreten Klassennamen der Sprachobjekte Bezug nehmen zu müssen. Auf diese Weise werden Abhängigkeiten zwischen Sprachelementklassen und ihren Klienten (d.h. den Klassen des administrativen Interpreterrahmens) vermieden, die diese Sprachelemente erzeugen. Für jede integrierte Prozesssprache steht eine Fabrik bereit, die die spezialisierten Sprachobjekte der Prozesssprache erzeugt. So werden z.B. SLANG-Transitionen als Spezialisierung einer atomaren Aktivität ausschließlich über die Methode `createAtomicActivity()` der SLANG-Fabrik erzeugt, die den Klassennamen kapselt. Die Verwen-

dung dieses Entwurfsmusters ergibt sich aus dem Umstand, dass bei der Integration einer neuen Prozesssprache die vom Interpreterrahmenwerk bereitgestellten Sprachelemente spezialisiert werden müssen, jedoch die spezialisierten Klassennamen vorab nicht bekannt sind. Die Klassennamen sind jedoch für die Instanziierung eines Sprachelements erforderlich, so dass sich hier eine (mit dem Entwurfsmuster vermeidbare) Abhängigkeit ergibt. Die Anwendung dieses Entwurfsmuster ermöglicht die sprachunabhängige Realisierung einer wiederverwendbaren Funktionalität. Auf diese Weise konnte z.B. der Ladevorgang, d.h. die Instanziierung eines Prozessfragments, generisch realisiert werden, da die einzelnen spezialisierten Sprachelemente unabhängig von einer Prozesssprache erzeugt werden.

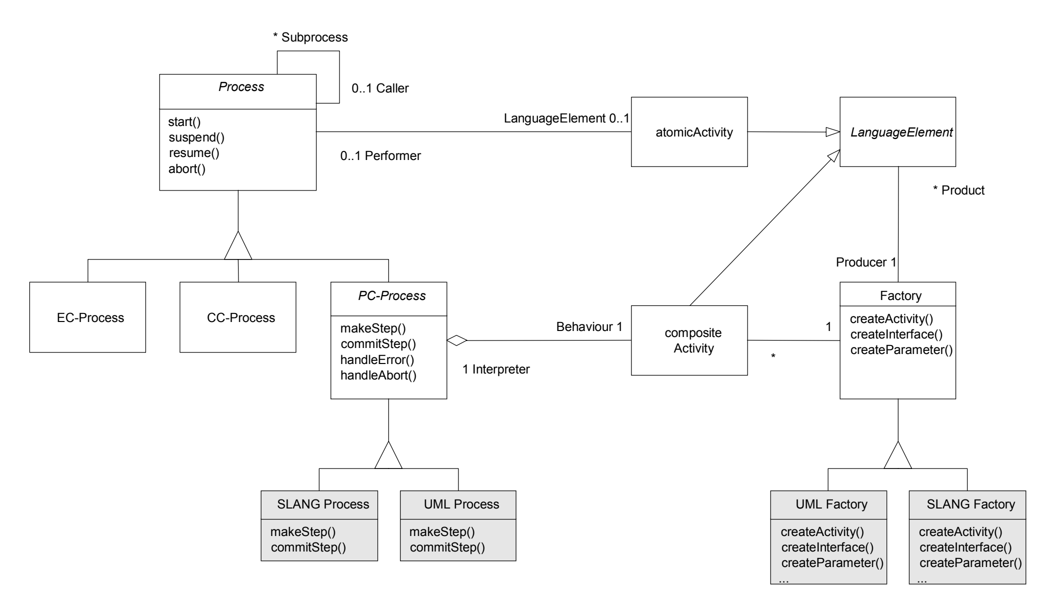


***Abb. 66**: Technische Klassen des Interpreterrahmenwerks*

Die zweite technische Klassenfamilie bilden die Subklassen der Klasse `Process`. Diese werden für die Ausführungssteuerung eines Kontextes benötigt, d.h. eine Instanz der Klassen `EC_Process`, `CC_Process` bzw. `PC_Process` operationalisiert jeweils einen Kontext in der PRIME-Werkzeugumgebung. Die drei Prozesstypen steuern je nach Kontextart die Kommunikation mit der Durchführungsdomäne (für die Aktivierung eines Ausführungs- oder Entscheidungskontexts) oder starten eine weitere Interpreterinstanz.

Die Schnittstelle der `Process`-Klasse bietet daher Funktionalität hinsichtlich der Prozessausführung an, die das Starten, Suspendieren oder Abbrechen des Prozesses steuert. Prozesse stehen in einer Aufrufhierarchie, da die Ausführung eines Prozesses zu weiteren Subprozessen führen kann. Während der Interpretation eines Plankontextes werden sukzessive Kontexte deduziert, die durch entsprechende `Process`-Instanzen operationalisiert werden. Die Klasse `PC_Process` als Interpreter eines Prozessfragments ist daher der Aufrufer von weiteren SubProzessen.

7.5.2.3 Integration einer neuen Sprache

Die Integration einer Sprache erfolgt, indem die sprachlichen und technischen Klassen spezialisiert und die spezialisierten Sprachelemente durch Überladen von Methoden ggf. angepasst werden. Die in Abb. 65 und Abb. 66 dargestellten Methoden stellen einen Auszug aus den Variationspunkten der Sprachelemente dar.

Die dort dargestellten Basisklassen realisieren bereits ein standardisiertes Verhalten, welches bei der Integration wiederverwendet oder angepasst wird. Über diese Variationspunkte hinaus realisieren die Klassen den Anschluss an die Schnittstellenschicht, der bei der Integration einer Sprache bzw. eines Interpreters in das GARPEM-Rahmenwerk zwingend erforderlich sind. Diese Dienste sind keine Variationspunkte, da sie weitgehend unabhängig von einer spezifischen Sprache sind. Dazu gehört z.B. die Anbindung an den PRIME-Kommunikationsmechanismus und die Datenbankanbindung.

Zur Klärung wollen wir abschließend noch einmal betonen, dass die Interpreterfunktionalität auf die sprachlichen und technischen Klassen der Sprachelementeschicht und der administrativen Interpreterschicht verteilt ist, so dass Interpreter hier nicht als eigenständige Entitäten innerhalb der GARPEM-Architektur auftreten. Die Verteilung der Funktionalität ist ein Effekt des objektorientierten Entwurfs, da hier die Konzepte der Domäne sowie deren Verantwortlichkeiten im Vordergrund stehen und keine funktionale Dekomposition des Systems vorgenommen wird. Der Interpretationsvorgang zieht daher alle Klassen der oberen beiden Schichten des GARPEM-Rahmenwerkes mit ein, da über die Schnittstellen die Verantwortlichkeiten der Klassen realisiert werden.

7.5.3 Kontrollmodell

Bei der Realisierung eines allgemeinen Rahmenwerkes für die Integration von verschiedensprachlichen Prozesssprachen-Interpretern ist der Kontrollfluss innerhalb des Rahmenwerkes von entscheidender Bedeutung. Das Interpreterrahmenwerk muss sowohl flexibel die unterschiedlichen Kontrollflussparadigmen der einzelnen Prozesssprachen berücksichtigen als auch genügend Unterstützung bieten, um Interpreter auf einfache Weise in das Rahmenwerk zu integrieren. Beim Entwurf ist somit das Spannungsfeld zwischen hoher Flexibilität bei der Integration von heterogenen Sprachen zu Lasten eines geringen Wiederverwendbarkeitsgrades und der breiten Unterstützung eines speziellen Kontrollflussparadigmas bei Verzicht auf die Allgemeinheit zu berücksichtigen.

7.5.3.1 Ausführungsmodell von Kontextkomponenten

Wie in Abschnitt 7.5.2.2 erläutert, wird die Ausführung eines AK-, EK- oder PK-Kontextkomponente von einer Instanz der Klasse `EC_Process`, `CC_Process` bzw. `PC_Process` gesteuert. Damit übergeordnete Dienste des administrativen Interpreterrahmens, die sich auf Ausführungsinformationen beziehen, unabhängig von der Kontextart und der zugrunde liegenden Prozesssprache realisiert werden können, ist eine Vereinheitlichung des Ausführungsmodells dieser Klassen erforderlich. Beispiele für Dienste, die zustandsbasierte Informationen benötigen, sind Komponenten zur Prozessausführungskontrolle und -überwachung sowie die Protokollkomponente für die Prozessspuraufzeichnung. Eine einheitliche Steuerung der Prozessausführung ist dabei bereits durch die Schnittstelle der Klasse `Process` für alle drei Kontextarten gewährleistet (vgl. Abb. 66). Hier bezieht sich die Vereinheitlichung jedoch nicht nur auf die Prozessschnittstelle, sondern auch auf die Ausführungszustände eines Prozesses.

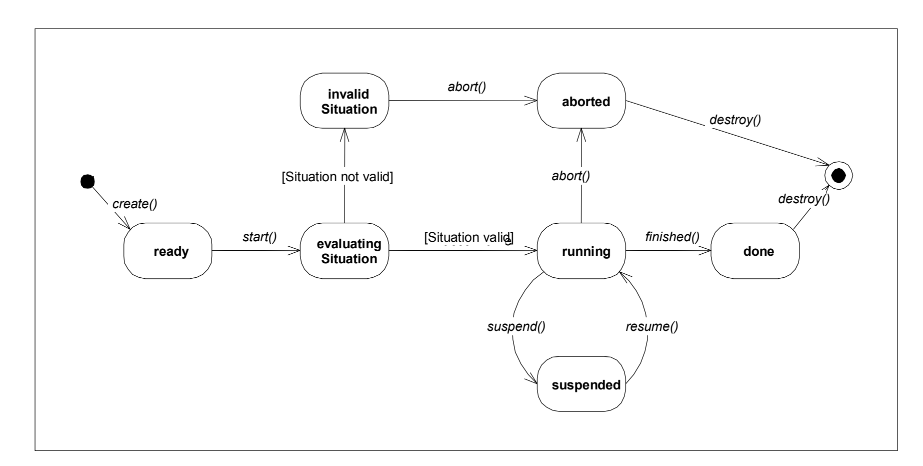


***Abb. 67**: Ausführungszustände der Klasse* `Process`

Das Zustandsdiagramm in Abb. 67 definiert das Ausführungsmodell der Klasse `Process` und orientiert sich an verbreiteten Zustandsmodellen für Workflows wie z.B. [JaBu96; BMCJ98]. Die abgebildeten Zustände werden von den spezialisierten Klassen `EC_Process`, `CC_Process` und `PC_Process` geteilt, so dass hier eine übergreifendes Ausführungsverhalten dargestellt wird. Um den unterschiedlichen Ausführungssemantiken der drei Kontextarten des Kontextmodells gerecht zu werden, werden die einzelnen Zustände je nach Kontextart in weitere Unterzustände verfeinert; die Verfeinerung des Zustandes `Running` für die Klassen `PC_Process` und `CC_Process` wird weiter unten dargelegt. Im Folgenden wird zunächst die Bedeutung der übergeordneten Zustände erläutert.

- `Ready`: Ein Prozess im Zustand `ready` ist instanziiert und kann ausgeführt werden. Zu diesem Zeitpunkt sind alle für die Ausführung relevanten Daten präsent. Im Falle von PC_Process ist das Fragment, welches das Verhalten des Prozesses definiert, bereits geladen und initialisiert. Im Falle von `EC_Process` und `CC_Process` sind die auszuführende Werkzeugaktion bzw. die Kontextalternativen des Entscheidungskontextes geladen und die entsprechende Werkzeuginstanz ausführungsbereit.

- `EvaluatingSituation`: Die aktuellen Situationsdaten des Prozesses werden gemäß dem Situationsausdruck ausgewertet. Der Situationsausdruck als Vorbedingung entscheidet darüber, ob mit der Prozessausführung fortgefahren oder abgebrochen wird.

- `InvalidSituation`: Dieser Zustand repräsentiert eine ungültige Situation, d.h. mit der Ausführung des Prozesses kann nicht fortgefahren werden.

- `Running`: Im Zustand `running` wird der Prozess von einem Agenten ausgeführt. Im Falle eines `PC_Process` wird das Fragment in der Leitdomäne interpretiert, im Falle eines `EC_Process` wird die entsprechende Werkzeugaktion ausgeführt und im Falle eines `CC_Process` die Benutzerauswahl zwischen den Kontextalternativen vollzogen.

- `Suspended`: Ein Prozess kann während der Ausführung unterbrochen werden und befindet sich dann im Zustand `suspended`.

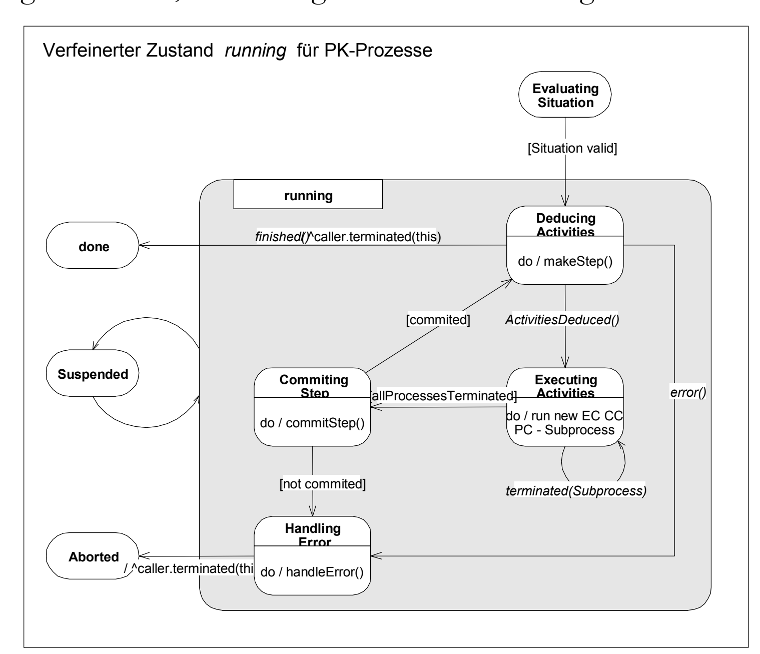
- `Aborted`: Im Gegensatz zum Zustand `suspended`, der eine Wiederaufnahme der Prozessausführung ermöglicht, repräsentiert der Zustand `aborted` einen unwiderruflichen Prozessabbruch. Dieser kann zum einen aufgrund eines Fehlverhaltens während der Ausführung eintreten oder z.B. durch den Benutzer eingeleitet werden.

- Done: Innerhalb des Zustandes done liegen die Ergebnisdaten der Prozessausführung vor und auf den Prozess kann von außen immer noch zugegriffen werden

Ausführungsmodell von PK-Komponenten

Die erläuterten Zustände legen ein übergeordnetes Ausführungsmodell von sowohl AK-, EK- als auch PK-Prozessen fest. Mit der Verfeinerung der Zustände wird das Ausführungsmodell an die drei Prozesstypen angepasst. Bei PK-Prozessen findet die Interpretation eines Fragmentes im Zustand running statt, dessen Verfeinerung in Abb. 68, rechts dargestellt ist und im Folgenden erläutert wird.

Abb. 68: Ausführungsmodell der Klasse PC_Process



Der Interpretationsvorgang eines Fragmentes erfolgt schrittweise und ist in drei Phasen eingeteilt, die durch die Zustände DeducingActivities, ExecutingActivities und CommitingStep gekennzeichnet sind. Innerhalb eines fehlerfreien Interpretationsschrittes werden alle drei Zustände nacheinander durchlaufen. Beim Eintritt in den Zustand running ist das Fragment (compositeActivity) bereits geladen und initialisiert, so dass der Interpretationsvorgang gestartet werden kann. Zunächst werden in der ersten Phase im Zustand DeducingAcitvities die Aktivitäten des Fragmentes bestimmt, die in einem Schritt auszuführen sind. Im Falle von rein sequenziellen Sprachen wird maximal eine Aktivität in einem Interpretationsschritt deduziert. Sprachen, die Parallelität zulassen, können in einem Schritt ggf. mehrere Aktivitäten bestimmen, die dann parallel ausgeführt werden. Welche Aktivitäten deduziert werden hängt dabei vom Kontrollfluss im Fragment ab. Da die Kontrollmodelle der Prozessmodellierungssprachen divergieren, wird dieser Zustand je nach Sprache unterschiedlich realisiert.

In einer zweiten Phase werden im Zustand ExecutingActivities die zuvor bestimmten Aktivitäten ausgeführt. Wie bereits erläutert, repräsentieren die Aktivitäten AK-, EK- oder PK-Kontextkomponenten, für deren Ausführung eine entsprechende Process-Instanz erforderlich ist. Für jede deduzierte Aktivität wird in diesem Zustand ein eigener Prozess abgespalten, der die Ausführung des entsprechenden PRIME-Kontextes vollzieht. Die abgespaltenen Prozesse sind dabei alle Subprozesse des aufrufenden PK-Prozesses und werden von diesem gesteuert.

Die Zustandsänderungen der Subprozesse werden stets an den aufrufenden PK-Prozess weitergeleitet, so dass dieser insbesondere über deren Termination informiert wird. Ein Subprozess terminiert entweder fehlerfrei, fehlerhaft oder wurde zeitweise unterbrochen, was sich in den oben erläuterten Zustände done, aborted bzw. suspended äußert. Die Termination eines Subprozesses stellt ein Ereignis dar, welches vom PK-Prozess registriert wird und im Folgezustand vom Interpreter bestätigt werden muss. Der Zustand ExecutingActivities wird verlassen, sobald alle abgespaltenen Subprozesse terminiert sind.

In der dritten Phase muss die Ausführung der abgespaltenen und terminierten Prozesse im Zustand CommittingStep bestätigt werden. Die Bestätigung ist erforderlich, da die abgespaltenen Subprozesse nicht zwingend fehlerfrei terminieren. Da die Ausnahmebehandlung je nach Prozesssprache unterschiedlich gehandhabt wird und zum Teil auch durch entsprechende Sprachkonzepte unterstützt wird, wird dem Prozesssprachen-Interpreter in diesem Zustand die Möglichkeit gegeben, geeignet auf die Termination der Subprozesse zu reagieren. Der Interpretationsschritt wird entweder bestätigt oder führt im Falle einer Ausnahme zu einem Abbruch der Interpretation.

Die drei Phasen werden solange wiederholt, bis vom Prozesssprachen-Interpreter signalisiert wird, dass die Interpretation des Fragmentes beendet ist. In diesem Falle verlässt der PK-Prozess den Zustand running und geht in den Zustand done über, der eine erfolgreiche Beendigung repräsentiert. Im Gegensatz dazu kann die fehlerhafte Termination des PK-Prozesses zwei Ursachen haben. Zum einen kann während eines Interpretationsschrittes ein Fehler im Fragment auftreten. Dazu gehören Fehler, die bei unsachgemäße Modellierung des Fragmentes auftreten wie z.B. laufzeitbedingte Typ- oder Typzuweisungsfehler. Diese treten im Zustand DeducingActivities auf und führen zunächst zu einer Fehlerbehandlung im Zustand HandlingError und dann zum Abbruch des PK-Prozesses. Zum anderen kann die Ausführung eines Prozesses im Zustand ExecutingActivities z.B. vom Benutzer abgebrochen werden. Wenn der Interpretationsschritt nicht im Zustand CommitStep bestätigt wird, führt dies ebenfalls zu einem Abbruch des PK-Prozesses.

Ausführungsmodell von AK- und EK-Prozessen

Analog zum Ausführungsmodell der Klasse PC_Process kann der running-Zustand von AK- und EK-Prozessen (Klassen EC_Process und CC_Process) verfeinert werden. Da die eigentliche Ausführung dieser Prozesskomponenten in der Durchführungsdomäne stattfindet, kann jedoch innerhalb des Interpreterframeworks von den Subständen des Zustands running abstrahiert werden, so dass wir hier nicht näher darauf eingehen[40].

Nachrichten und Variationspunkte

Zwischen dem Ausführungsmodell eines Prozesses (vgl. Abb. 67) und den Variationspunkten des Interpreterrahmenwerkes besteht ein enger Zusammenhang.

[40] Die Phasen der Ausführungs- und Entscheidungskontextinterpretation im Zustand running wurden bereits in Abschnitt 7.3.2.5 bei der Darstellung des ContextManager-Pakets beschrieben; Details zur formalen Zustandsmodellierung sind in [Schm99] zu finden.

Durch Offenlegung der Ausführungszustände eines Prozesses können Ereignisse der Prozessausführung an den spezialisierten Prozesssprachen-Interpreter weitergeleitet werden. Dabei entspricht jeder Zustand und Zustandsübergang eines Prozesses einem Nachrichtentyp, der innerhalb des Interpreterrahmenwerkes versendet wird. Die Prozesssprachen-Interpreter sind für die Weiterverarbeitung des Nachrichtentyps verantwortlich und können darauf entsprechend reagieren, indem für jede Nachricht ein Prozesssprachen-spezifisches Verhalten implementiert wird. Die Empfänger der Nachrichtentypen sind die technischen und sprachlichen Klassen, für die jeweils ein spezialisierter Variationspunkt (virtuelle Methode) existiert, der das Verhalten der Klasse auf den empfangenen Nachrichten festlegt (siehe Tab. 16). Die Nachrichtentypen der Prozessausführung werden für die Kontrollinversion im Interpreterrahmenwerk genutzt, so dass sich der Kontrollfluss im Interpreterrahmenwerk am Prozessausführungszustand orientiert.

***Tab. 16**: Zuordnung der zustandsbasierten Nachrichtentypen der Prozessfragment-Interpretation zu den Variationspunkten der Interpreterklassen*

Frameworkelement / Nachrichtentyp	**Process**	**Composite Activity**	**Interface**	**Atomic Activity**	**Data Container**
Zustandsübergang					
Create()	Init()	Load(), Init()	ReadData()	Creating()	InitValue(), ReadValue()
Start()				Starting()	
[Situation valid]			AfterRead()	ValidSituation()	
[Situation not valid]				InvalidSituation()	
Abort()	handleAbort()			Aborting()	
Suspend()				Suspending()	
Resume()				Resuming()	
Finished()			Prepare-Write(), Write()	Finished()	WriteValue()
Destroy()	Destroy()	Destroy()	Destroy()	Destroy()	Destroy()
Zustand					
Ready				Ready()	
EvaluatingSituation				EvaluatingSituation()	
Running	MakeStep() commitStep() handleError()			Running()	
Done				Done()	
Suspended				Suspended()	
Aborted				Aborted()	
InvalidSituation				InvalidSituation()	

7.5.4 Verwandte Ansätze

In der Literatur finden sich nur wenige Ansätze, die allgemeine objektorientierte Frameworks für die Entwicklung von Interpretern bereitstellen. Nach [HaHK97] ist dies darauf zurückzuführen, dass Architekturen für die Sprachimplementierung, d.h. Übersetzer, sich eher an den einzelnen Übersetzungsphasen orientieren als an den Sprachelementen selbst. Ausnahmen bilden das TaLE-Framework [HaHK97], das Etyma-Framework [BaLi96] und ein in [KoMö95] beschriebenes Framework für Interpreter von numerischen Ausdrücken.

Diese Interpreterrahmenwerke bilden die Sprachstruktur, d.h. die Syntax, vollständig oder teilweise auf Klassen ab. Diese Vorgehensweise ist naheliegend, da Syntax und Klassenstruktur auf gleiche Weise ein (Sprach-)Schema vorgeben, dessen Instanzen gerade die Sätze einer Sprache sind, d.h. aus einer Menge von assoziierten Objekten bestehen. Die Semantik eines Sprachelements wird durch

Implementierung des Klassenkörpers definiert und kann mit Hilfe der Spezialisierungsbeziehung modifiziert werden. Die abstrakten Schnittstellen der Sprachelemente entsprechen dabei den Variationspunkten des Interpreterrahmenwerkes. Der Grundgedanke, die einzelnen Sprachelemente durch Klassen zu repräsentieren, die durch Spezialisierung angepasst werden, wurde bei der Entwicklung des GARPEM-Interpreterframeworks übernommen.

7.5.5 Zusammenfassung

In diesem Abschnitt wurde das Interpreterrahmenwerk GARPEM für die Integration von Prozesssprachen-Interpretern in die PRIME-Umgebung erarbeitet. Die wiederverwendbaren Teilkomponenten des Rahmenwerkes wurden identifiziert, die durch Spezialisierung und Erweiterung an die spezifischen Bedürfnisse einer Prozesssprache angepasst werden können. Das in Kapitel 6 entwickelte PSM2-Modell, welches die Sprachelemente einer Prozesssprache identifiziert, die für eine komponentenbasierte Integration erforderlich sind, wurde direkt auf die Klassenstruktur des Interpreterrahmenwerkes abgebildet und stellt die Basis für die sprachlichen Teilkomponenten dar. Auf diese Weise ist gewährleistet, dass Sprachen, die sich für eine Integration eignen, auch einfach integriert werden können und vom Rahmenwerk unterstützt werden.

Die Kontrollinversion basiert auf den Ausführungszuständen eines Prozesses, mit dem deduzierte Kontexte in der PRIME-Umgebung operationalisiert werden. Für jeden Zustand und Zustandsübergang existiert ein Nachrichtentyp, der innerhalb des Rahmenwerkes versandt wird. Dabei werden zustandsbasierte Ausführungsinformationen im Rahmenwerk offengelegt, so dass ein spezialisierter Prozesssprachen-Interpreter ein sprachspezifisches Verhalten realisieren kann.

Bislang wurde innerhalb des GARPEM-Frameworks Interpreter für SLANG-Netze und UML-Statecharts integriert. Der Wiederverwendungsgrad betrug dabei 85 % bzw. 87 %. Außerdem gelang es, eine frühere Version der Prozessmaschine, in der Prozessfragmente als C++-Methoden formuliert wurden, nahtlos in das GARPEM-Framework zu integrieren.

7.6 Fazit

Gegenstand dieses Kapitels war die PRIME-Architektur für prozessintegrierte Werkzeug-Umgebungen. PRIME setzt die in Kapitel 5 und 6 dargestellten Konzepte zur integrierten Werkzeug- und Prozessmodellierung in Form zweier generischer, objektorientierter Frameworks um. In der Durchführungsdomäne definiert das GARPIT-Framework die Grundstruktur eines prozessintegrierten Werkzeugs. Es stellt vorgefertigte Komponenten zur Synchronisation mit der Leitdomäne und zur Interpretation des Umgebungsmodells bereit und erlaubt die flexible Erweiterung um spezifische Werkzeugfunktionalitäten. Für die Integration existierender Werkzeuge wurde das GARPIT-Framework zu einem generischen Prozessintegrations-Wrapper modifiziert und anhand der Integration von insgesamt drei Fremdwerkzeugen validiert. In der Leitdomäne steht mit dem GARPEM-Framework ein generischer Rahmen für die Integration spezifischer Prozessspracheninterpreter zur Verfügung. Das GARPEM-Framework wurde am Beispiel der Einbettung eines SLANG- und eines UML-Statechart-Interpreters validiert.

Beide Sub-Frameworks des PRIME-Ansatzes sind durch die gemeinsam genutzten Anteile des Umgebungsmodells sowie über das in diesem Kapitel beschriebene Interaktionsprotokoll aufeinander abgestimmt. Somit ist die komplette Integration zwischen den Prozessdomänen bereits auf Framework-Ebene vorweggenommen. Der Umgebungsentwickler braucht sich somit nicht mehr um die schwierige architekturelle und technische Integration zwischen Modellierungs-, Leit- und Durchführungsdomäne zu kümmern, sondern muss lediglich die Frameworks an den dafür vorgesehenen Variationspunkten um spezifische Funktionalität anreichern. Eine Reihe von vordefinierten Entwicklungs- und Metamodellierungswerkzeugen erleichtern ihm die Arbeit zusätzlich.



Anwendungen

Der PRIME-Ansatz wurde bei der Entwicklung und Prozessintegration mehrerer Werkzeug-Umgebungen erfolgreich eingesetzt und validiert. In diesem Kapitel geben wir zunächst einen kurzen Überblick über die Entwicklungshistorie des PRIME-Rahmenwerks und seiner Anwendungen (Abschnitt 8.1) und gehen dann genauer auf die verfahrenstechnische Entwurfsumgebung PRIME-IMPROVE ein (Abschnitt 8.2). Eine Beispielsitzung illustriert den Umgang mit einer PRIME-basierten Umgebung aus Sicht des Methodeningenieurs und des Applikationsingenieurs (Abschnitt 8.3).

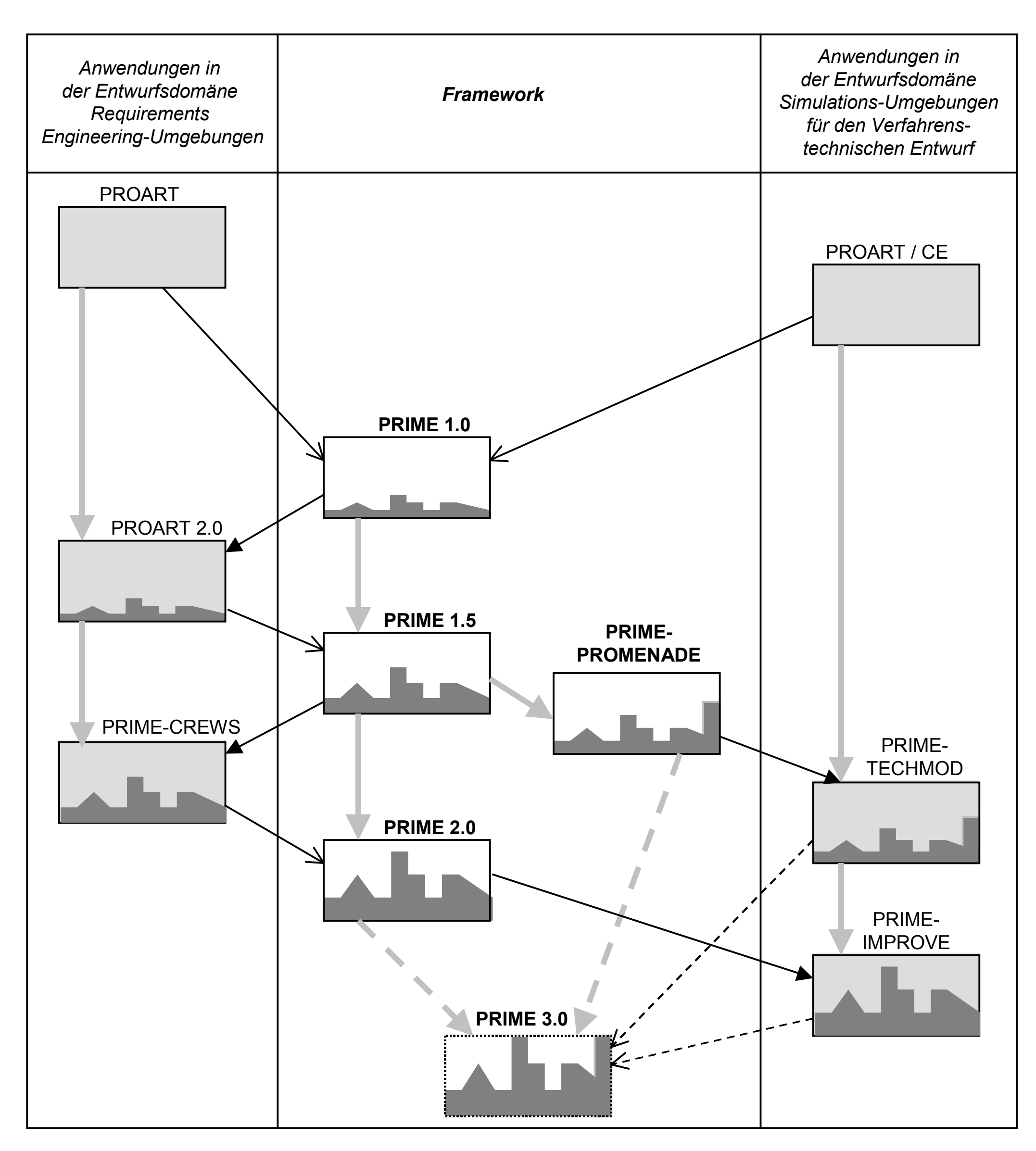
8.1 Entwicklungshistorie

Übereinstimmend wird in der Literatur die Framework-Entwicklung als ein kontinuierlicher und iterativer Prozess angesehen [FaSc99; Grif98; Gry*99]. Innerhalb eines Anwendungsbereichs werden ausgehend von einer repräsentativen Applikation schrittweise Abstraktionen gebildet, die in einem Framework generalisiert und für neue Applikationen wiederverwendet werden können. Auf Basis der mit der Framework-Anwendung gesammelten Erfahrungen können dann weitere Gemeinsamkeiten sowie zusätzlich benötigte Basisfunktionalitäten identifiziert werden, die dann als verallgemeinerte Softwarebausteine heraus faktorisiert und in die nächste Entwicklungsstufe des Frameworks aufgenommen werden.

Framework-Entwicklung als iterativer Prozess

Auch das PRIME-Framework ist das (vorläufige) Resultat eines solchen Reifeprozesses, der sich über mehrere Iterationen erstreckt hat und in dem sukzessive der Anteil der wiederverwendbaren Komponenten und die Funktionalität des Frameworks erhöht wurden. Um die unterschiedlichen Anwendungen des PRIME-Frameworks, die in diesem Kapitel sowie in [Pohl96; Döm*96; DöPo98; Dömg99; HaPW98; Haum00; Poh*99; WeBa99; JaLW99] genauer beschrieben werden, besser in einen Gesamtzusammenhang einordnen zu können, geben wir im Folgenden einen kurzen Abriss über die Entwicklungshistorie des PRIME-Frameworks und seiner Anwendungen. Der zunehmende Reifegrad des PRIME-Frameworks wird in Abb. 69 durch die immer größer werdenden dunkelgrauen Flächen (Framework-Anteile) der einzelnen Framework-Versionen symbolisiert.

Abb. 69: *Entwicklungshistorie des PRIME-Rahmenwerks*



PROART und PROART/CE

Den Ausgangspunkt für die erste Version des PRIME-Rahmenwerks bildete die Requirements Engineering-Umgebung *PROART*[41] [Pohl96], die im Rahmen des EU-Grundlagenprojekts NATURE [JRSD99] entwickelt wurde. Das Hauptaugenmerk dieser Umgebung lag auf der Nachvollziehbarkeit von Requirements Engineering-Prozessen auf der Grundlage einer automatisierten, prozessbegleitenden Aufzeichnung von feingranularen Abhängigkeiten zwischen Entwurfsprodukten, die in den unterschiedlichen PROART-Werkzeugen bearbeitet werden. In einer interdisziplinären Kooperation mit Lehrstuhl für Prozesstechnik der RWTH Aachen wurde erkannt, dass sich die aus dem Requirements Engineering bekannten Konzepte zur Nachvollziehbarkeitsunterstützung auch auf den rechnergestützten, konzeptuellen Entwurf verfahrenstechnischer Anlagen übertragen lassen [JaMa96]. Dies wurde anhand der Umgebung PROART/CE[42], einer um zusätzli-

---

[41] PROART: **P**rocess and **R**ep**O**sitory based **A**pproach for **R**equirements **T**raceability

[42] PROART/CE: **P**rocess and **R**ep**O**sitory based **A**pproach for **R**equirements **T**raceability / **C**hemical **E**ngineering

che verfahrenstechnische Werkzeuge erweiterte Version der PROART-Umgebung, demonstriert [Döm*96].

Bereits die Werkzeuge der PROART- bzw. PROART/CE-Umgebung waren in der Lage, sich dynamisch an den aktuellen Prozesszustand anzupassen. Das diesem Verhalten zugrunde liegende Prozesswissen lag jedoch nicht in Form expliziter Prozessmodelle vor, sondern manifestierte sich in hartverdrahteten, über die einzelnen Werkzeuge verteilten Code-Abschnitten. Entsprechend aufwändig gestaltete sich die Umsetzung von neuen Prozessen oder die Änderung existierender Prozesse. Das PRIME-Framework in der Version 1.0 setzte erstmals den in Kapitel 5 beschriebenen integrierten Prozess- und Werkzeugmodellierungsansatz um und vereinheitlichte darüber hinaus die zwischen den ursprünglichen PRO-ART-Werkzeugen noch sehr heterogene Architektur in Hinblick auf Datenbank-, Benutzerschnittstellen- und Kommunikationsanbindung. Erfolgreich validiert wurde das PRIME 1.0-Framework mit der Re-Implementierung der Werkzeuge der PROART-Umgebung. Im Vergleich zur Vorgängerumgebung zeichnete sich die PROART 2.0-Umgebung aus Entwicklersicht durch eine stark gesteigerte Wiederverwendung, einen damit verbundenen verminderten Entwicklungsaufwand und eine wesentlich größere Änderungsfreundlichkeit aus. In der Anwendung machte sich das vereinheitlichte „Look and Feel“ der Werkzeuge in einer stark verbesserten Benutzbarkeit positiv bemerkbar. Darüber hinaus war eine signifikant höhere Stabilität und Performanz der Umgebung zu beobachten. Dies war darauf zurückzuführen, dass kritische Komponenten nicht redundant in allen Werkzeuge jeweils spezifisch implementiert wurden, sondern als allgemein verwendbare Framework-Komponenten mit besonderer Sorgfalt entworfen, realisiert und getestet wurden.

PRIME 1.0: Umsetzung der in Kapitel 5 beschriebenen Prozess- und Werkzeugmodellierung

PRO-ART 2.0

Basierend auf den Erfahrungen mit der PRO-ART 2.0-Entwicklung wurde in der nächsten Iterationsstufe des PRIME-Frameworks (Version 1.5) der Schwerpunkt zunächst auf eine bessere Unterstützung der Prozess- und Werkzeugmodellierung durch benutzerfreundliche Metamodellierungswerkzeuge gelegt. Zudem wurde das PRIME-Framework von Unix (Solaris) auf die Windows NT-Plattform portiert, um eine Anbindung an das GUI-Framework MFC (Microsoft Foundation Classes) zu ermöglichen. Dies geschah primär in Hinblick auf die im Rahmen des EU-Projekts CREWS[43] zu entwickelnden Werkzeuge für die Ableitung von konzeptuellen Modellen aus multimedial erfassten Anwendungsszenarien. Hier versprach MFC eine bessere technische Unterstützung für Multimedia-Inhalte (Video, Bilder, Audio etc.) als das bislang von uns verwendete GUI-Toolkit ILOG Views und andere unter Unix frei verfügbare Multimedia-Bibliotheken. Die auf Basis von PRIME 1.5 entwickelte PRIME-CREWS-Umgebung erweitert die Funktionalität von PRO-ART 2.0 um einen multimedialen Whiteboard-Editor, einen Zielmodell-Editor, einen Message Sequence Chart-Editor und ein Werkzeug zur Unterstützung von Review-Prozessen. Einzelheiten zur PRIME-CREWS-Umgebung sind in [HaPW98; Haum00] zu finden.

PRIME 1.5: Erweiterung um Metamodellierungswerkzeuge, Portierung auf Windows NT, Anbindung von Multimedia-Komponenten

PRIME-CREWS

[43] CREWS: Cooperative Requirements Engineering With Scenarios

PRIME-PROMENADE: Projektspezifische Filterung von Nachvollziehbarkeitsinformationen

Im Rahmen der Dissertation von Ralf Dömges wurden Erweiterungen in Richtung einer flexiblen Aufzeichnung von Nachvollziehbarkeitsinformationen mithilfe eines projektspezifisch anpassbaren Filtermechanismus vorgenommen. Diese *PRIME-PROMENADE*[44] genannte Framework-Erweiterung wurde bei der Entwicklung der verfahrenstechnischen Umgebung *PRIME-TECHMOD*[45] eingesetzt und validiert. Nähere Informationen sind in [Dömg99; DöPo98; DöPS98] zu finden. Die Arbeiten an PRIME-PROMENADE bzw. PRIME-TECHMOD erfolgten aus organisatorischen Gründen innerhalb eines eigenen Entwicklungszweigs unabhängig von der weiter unten beschriebenen Weiterentwicklung des PRIME-Frameworks. Eine Zusammenführung dieses Entwicklungsstrangs mit der Version 2.0 des PRIME-Frameworks (siehe unten) steht zurzeit noch aus.

PRIME 2.0: Integration von Fremdwerkzeugen, Interoperabilität zwischen Prozesssprachen, Erfassung und Visualisierung von Prozessspuren

Weiter voran getrieben wurde die Weiterentwicklung des PRIME-Frameworks im Rahmen des interdisziplinären Sonderforschungsbereichs *IMPROVE*[46] an der RWTH Aachen, in dem Informatiker, Verfahrenstechnik- und Kunststofftechnik-Ingenieure seit 1997 an einer umfassenden Unterstützungsumgebung für den verfahrenstechnischen Entwurf arbeiten. Wesentliche Neuerungen der Version 2.0 des PRIME-Frameworks sind das auf der generischen Werkzeugarchitektur basierende Integrationskonzept für die Einbindung von *Fremdwerkzeugen* (vgl. Abschnitt 7.4), die Integration und Interoperabilität von Prozesssprachen zur Plankontextdefinition gemäß dem in Kapitel 6 vorgestellten Sprachintegrationsansatz sowie eine zentralisierte Erfassung und Visualisierung von Prozessspuren. Im Rahmen der Kooperation mit anderen Projektpartnern wurde das PRIME-Framework darüber hinaus mit einer Reihe von zumeist CORBA-basierten Schnittstellen zu externen Komponenten versehen. Auf Basis des PRIME 2.0-Frameworks wurde die PRIME-IMPROVE-Umgebung entwickelt, die wir im Folgenden im Detail vorstellen werden.

8.2 PRIME-IMPROVE

8.2.1 Überblick über SFB IMPROVE

Unter einem *verfahrenstechnischen Prozess* versteht man die Verknüpfung physikalischer, chemischer, biologischer und informationstechnischer Vorgänge, um Ausgangsstoffe nach Art, Eigenschaft und Zusammensetzung so zu verändern, dass ein gewünschtes stoffliches Produkt entsteht. Dieser Prozess läuft auf einer Anlage ab. Der Entwurf oder die Modifikation eines verfahrenstechnischen Prozesses (im Folgenden *VT-Prozess* genannt) und seiner anlagentechnischen Umsetzung ist Ziel des *Modellierungsprozesses* (im Folgenden *M-Prozess* genannt[47]). Ausgehend von einer

---

[44] PROMENADE: **PR**ojektspezifische, **ME**thodische **N**achvollziehbarkeit von **A**nfer**DE**rungsspezifikationen

[45] TECHMOD: **T**raced **E**ngineering of **CH**emical process **MOD**els

[46] IMPROVE: **I**nfor**m**atische Unterstützung übergreifender Entwicklungs**pro**zesse in der **Ve**rfahrenstechnik

[47] Die sprachliche Unterscheidung zwischen VT- und M-Prozess mag im Text etwas umständlich erscheinen, hat sich aber in der interdisziplinären Kooperation mit Verfahrenstechnik-Ingenieuren zur Vermeidung von Missverständnissen als unverzichtbar erwiesen.

groben Problemstellung wird im Team eine vollständige Spezifikation von VT-Prozess und Anlage erarbeitet, die damit das Produkt des M-Prozesses darstellt.

Der seit August 1997 von der Deutschen Forschungsgemeinschaft geförderte SFB IMPROVE entwickelt neuartige, informatische Konzepte, die die Verbesserung kooperativer M-Prozesse in der Verfahrenstechnik aus vier unterschiedlichen Blickwinkeln betrachten [NaWe99]: (1) *Direkte M-Prozessunterstützung* durch Beobachtung von Abläufen bei Entwicklern und Anbieten von für gut befundenen Abläufen zur Wiederverwendung; (2) *indirekte M-Prozessunterstützung* durch Sicherung von Struktur- und Konsistenzbedingungen der entstehenden komplexen Produkte mit Hilfe von Integrationswerkzeugen; (3) i*nformelle, multimediale Kooperation* der M-Prozessbeteiligten; (4) Projektkoordination durch ein *reaktives Administrationssystem*.

Die hier beschriebene PRIME-IMPROVE-Umgebung wurde im Rahmen des Teilprojekts *Direkte M-Prozessunterstützung* entwickelt, welches sich auf die Erfassung, Formalisierung und Steuerung arbeitsplatzbezogener Entwurfsvorgänge durch feingranulare, werkzeugbezogene M-Prozessfragmente konzentriert.

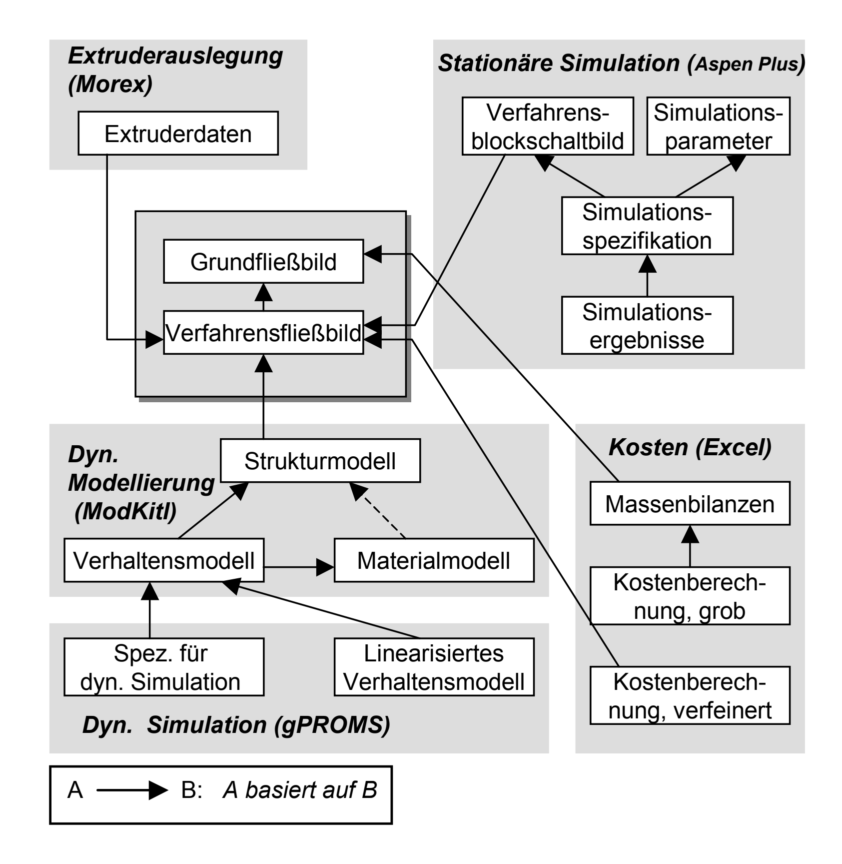
8.2.2 Werkzeuge der PRIME-IMPROVE-Umgebung

PRIME-IMPROVE integriert eine Reihe von Werkzeugen für die Modellierung, Analyse, Simulation und Dokumentation verfahrenstechnischer Prozesse. Zu diesen gehören die allgemein verwendbaren Werkzeuge des PRIME-Rahmenwerks (Abhängigkeitseditor, Hypertexteditor, Entscheidungseditor, Prozessspurenvisualisierer, Anleitungswerkzeug; vgl. Abschnitt 7.1.2), die kommerziellen Simulations- und Kalkulationswerkzeuge Aspen Plus, MS Excel, gPROMS und Morex sowie ein auf Basis des CAD-Werkzeugs Visio entwickelter Fließbildeditor, den wir aufgrund seiner besonderen Bedeutung für verfahrenstechnische M-Prozesse im Folgenden genauer beschreiben.

8.2.2.1 Stellung des Fließbilds im Gesamtprozess

Im Zuge der Verfahrensentwicklung entsteht eine Fülle von Informationseinheiten, die ständig weiter gepflegt und zugriffsbereit gehalten werden müssen. Eine besondere Rolle spielen dabei grafische Darstellungen in Form von *Fließbildern*, die Aufbau und Funktion der verfahrenstechnischen Anlage auf unterschiedlichen Abstraktionsniveaus wiedergeben [Doug88; Blas97]. Um das Fließbild herum lassen sich in natürlicher Weise andere wichtige Informationseinheiten gruppieren, die während des Entwurfsprozesses entstehen, z.B. Simulationsergebnisse, Kostenschätzungen, Entscheidungsdokumentationen und sicherheitstechnische Analysen. Abb. 70 illustriert die vielfältigen Querbezüge des Fließbilds zu anderen Informationseinheiten innerhalb des vom IMPROVE-Projekt betrachteten Anwendungsszenarios. Aufgrund der zentralen Stellung des Fließbilds als Kristallisationspunkt vielfältiger Entwurfsaktivitäten, die durch Praxisbeobachtungen [Jar*98a] untermauert wird, kann ein Werkzeug zur Fließbilderstellung besonders von der Assistenzfunktion der M-Prozessintegration profitieren.

Abb. 70:
Stellung des Fließbilds im Entwurfsprozess



8.2.2.2 Anforderungen an das Fließbildwerkzeug

Aus verfahrenstechnischer Sicht ergeben sich zwei zentrale Anforderungen an ein Werkzeug für den Fließbildentwurf: der Umgang mit komplexen Verfeinerungsstrukturen und die Organisation von Fließbildbausteinen innerhalb eines reichhaltigen Typsystems.

Komplexe Verfeinerungsstrukturen

Fließbilder werden über unterschiedliche Abstraktionsniveaus hinweg verfeinert und in jeweils spezifischen Repräsentationen dargestellt. Im Rahmen des SFBs haben wir uns besonders für zwei der drei in DIN 28004 genormten Fließbildvarianten interessiert: das *Grund-* und das *Verfahrensfließbild*[48].

Das Grundfließbild stellt die Gesamtfunktion des VT-Prozesses dar. Diese kann durch Aufteilung in komplex aufgebaute Teilfunktionen (VT-Prozessschritte) und atomare Grundoperationen (Unit Operations) sowie das Festlegen von Verknüpfungen zwischen Teilfunktionen und Operationen hierarchisch strukturiert werden. Im Grundfließbild steht die *Funktion* eines VT-Prozesselements im Vordergrund (z.B. Reaktion, Trennung). Es wird jedoch noch nicht über die apparatetechnische Realisierung eines Funktionselements entschieden. In der grafischen Darstellung verwendet man rechteckige Kästen für die Funktionen und Grundoperationen und gerichtete Kanten für die verbindenden Stoffströme.

[48] Darüber hinaus existiert das *Rohr- und Instrumentenfließbild*, das jedoch erst zur detaillierten Auslegung der Anlagenelemente und zur Spezifikation regelungstechnischer Sachverhalte eingesetzt wird und somit innerhalb der vom SFB betrachteten frühen Entwurfsphase nur eine nachgeordnete Rolle spielt.

Das Verfahrensfließbild dient der Darstellung der verfahrenstechnischen Anlage auf der Geräteebene. Teilfunktionen der Funktionsstruktur werden durch geeignete Apparate und Maschinen realisiert, in denen chemische, physikalische oder biologische Wirkungsabläufe stattfinden. In der grafischen Darstellung werden abstrahierte Symbole der für die Realisierung vorgesehenen Geräte verwendet.

Abb. 71 illustriert die schrittweise Ausgestaltung der VT-Prozessstruktur bis hin zur groben anlagentechnischen Umsetzung. Der initiale VT-*Prozess* wird zunächst um eine *Rückführung* von Stoffströmen angereichert und dann in die Teilfunktionen (VT-Process Steps) *Reaktion* und *Trennung* zerlegt. Die Trennung wird spezialisiert in die Grundoperation (Unit Operation) *Flüssigtrennung*, die auf der funktionalen Ebene nicht weiter verfeinert wird. Umgesetzt wird der VT-Prozessschritt *Reaktion* durch einen *Rührkesselreaktor* (linkes Symbol in der unteren Ebene von Abb. 71), während die *Flüssigtrennung* mithilfe einer *Destillationskolonne* realisiert wird (rechtes Symbol in der unteren Ebene von Abb. 71).

Man erkennt bereits an diesem kleinen Beispiel, dass rasch komplexe Verfeinerungsstrukturen entstehen, die durch vielfältige Verfeinerungsbeziehungen zwischen Fließbildausschnitten charakterisiert sind (Dekomposition, Anreicherung, Spezialisierung, Realisierung). Für eine „korrekte" Verfeinerung sind eine Reihe von Konsistenzbedingungen zu beachten, z.B. Balancierungsregeln für die ein- und ausgehenden Ströme oder Typkonsistenzen zwischen verfeinerten und verfeinernden Fließbildelementen. Beispielsweise würde es keinen Sinn machen, einen Teilfunktion *Reaktion* durch eine *Destillationskolonne* zu realisieren.

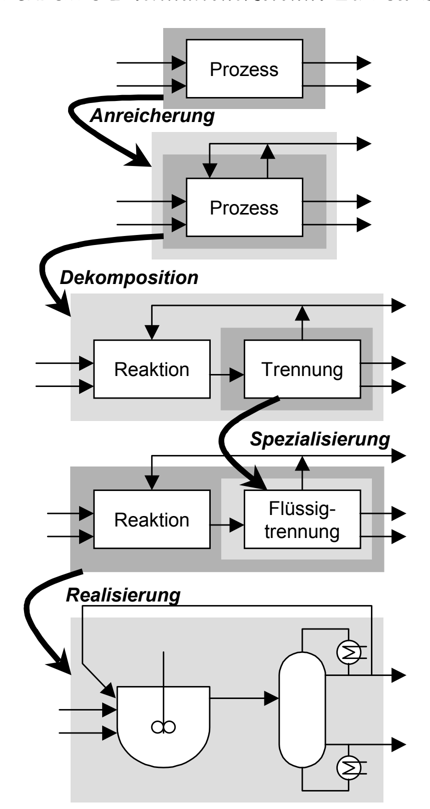


***Abb. 71**: Komplexe Fließbildverfeinerung*

Reichhaltiges, erweiterbares Typsystem für Fließbildbausteine

Ein ausdrucksstarkes Typsystem für Fließbildbausteine stellt ein unverzichtbares Hilfsmittel zur Steuerung der konsistenten Verfeinerung von Fließbildausschnitten

dar. Ein solches Typsystem muss Kompatibilitäten zwischen Bausteintypen entlang mehrerer Dimensionen ausweisen (z.B. dass ein U eine Spezialisierung von V darstellt oder dass X als Teil einer Dekomposition von Y auftreten kann) und eine semantisch reichhaltige Charakterisierung einzelner Bausteintypen erlauben (z.B. dass eine Destillationskolonne als Trenngerät nur dann in Frage kommt, wenn sich die Siedepunkte der zu trennenden Stoffe um mehr als 10°C unterscheiden).

Das Typsystem muss anpassbar und erweiterbar sein, da sich das Wissen über Bausteintypen und ihre Eigenschaften im ständigen Wandel befindet. Außerdem soll der Verfahrenstechniker bei der Definition eigener Bausteintypen unterstützt werden, um deren Wiederverwendung in anderen Anwendungskontexten zu forcieren. Benutzerdefinierte Bausteintypen entstehen z.B. durch Parametrierung generischer Bausteine oder Aggregation aus einfacheren Bausteintypen (z.B. die Hintereinanderschaltung mehrerer Kompressoren und Wärmetauscher).

8.2.2.3 Realisierung

Eine Evaluierung kommerzieller Werkzeuge für den Fließbildentwurf ergab, dass keines der verfügbaren Produkte die beschriebenen anwendungsseitigen Anforderungen auch nur annähernd erfüllte. Infolgedessen avancierte die Existenz offener Schnittstellen, die neben der M-Prozessintegration auch die Umsetzung der o.g. Anforderungen ermöglichen, zum entscheidenden Kriterium bei der Auswahl des zu integrierenden Werkzeugs. Die Wahl fiel schließlich auf Visio, dessen Stärken in der umfassenden und anpassbaren Symbolbibliothek und den auf COM-Schnittstellen basierenden Erweiterungsmechanismen liegen, die den feingranularen Zugriff auf Visios internes Objektmodell durch externe Zusatzkomponenten erlauben.

Die Anbindung von Visio an die PRIME-basierte M-Prozessunterstützung mithilfe des M-Prozessintegrations-Wrappers wurde bereits in Abschnitt 7.4.3 ausführlich beschrieben. Wir konzentrieren uns daher an dieser Stelle auf die Erweiterung von Visio um ein verfahrenstechnisches Fließbild-Datenmodell, das die oben skizzierten Anforderungen erfüllt.

Inhaltsorientiertes Produktmodell

In Visio werden technische 2D-Zeichnungen wie Fließbilder, Schaltbilder oder Diagramme gemäß einem sehr einfachen Objektmodell als flache Graphen bestehend aus ungetypten Knoten und Kanten verwaltet. Dieses Modell repräsentiert eine dokumentenorientierte Sichtweise, bei der Präsentationsaspekte (Layoutinformationen) im Vordergrund stehen, ohne dass eine inhaltliche Strukturierung der zu verwaltenden Information erfolgt.

Die im vorigen Abschnitt skizzierten Anforderungen an einen Fließbild-Datenmodell können mithilfe des Visio-Modells nicht umgesetzt werden, da es beispielsweise nicht ohne Weiteres möglich ist, Zeichenelemente mit semantisch reichhaltigen Typinformationen zu versehen, sie nach logischen Gesichtspunkten zu gruppieren und in einer Verfeinerungshierarchie anzuordnen sowie Konsistenzregeln durchzusetzen. Aus diesem Grund wurde das Visio-Objektmodell um ein Modell erweitert, das sich an der inhaltlichen Verfahrens- und Anlagenstruktur orientiert und einzelne Fließbilder (in Visio-Terminologie: Zeichenblätter) als Sichten auf das inhaltliche Modell betrachtet. Grundlage dieses Modells ist ein

allgemeines Domänenmodell für den verfahrenstechnischen Entwurf [BaMa98], das in Zusammenarbeit mit dem Lehrstuhl für Prozesstechnik an der RWTH Aachen überarbeitet wurde.

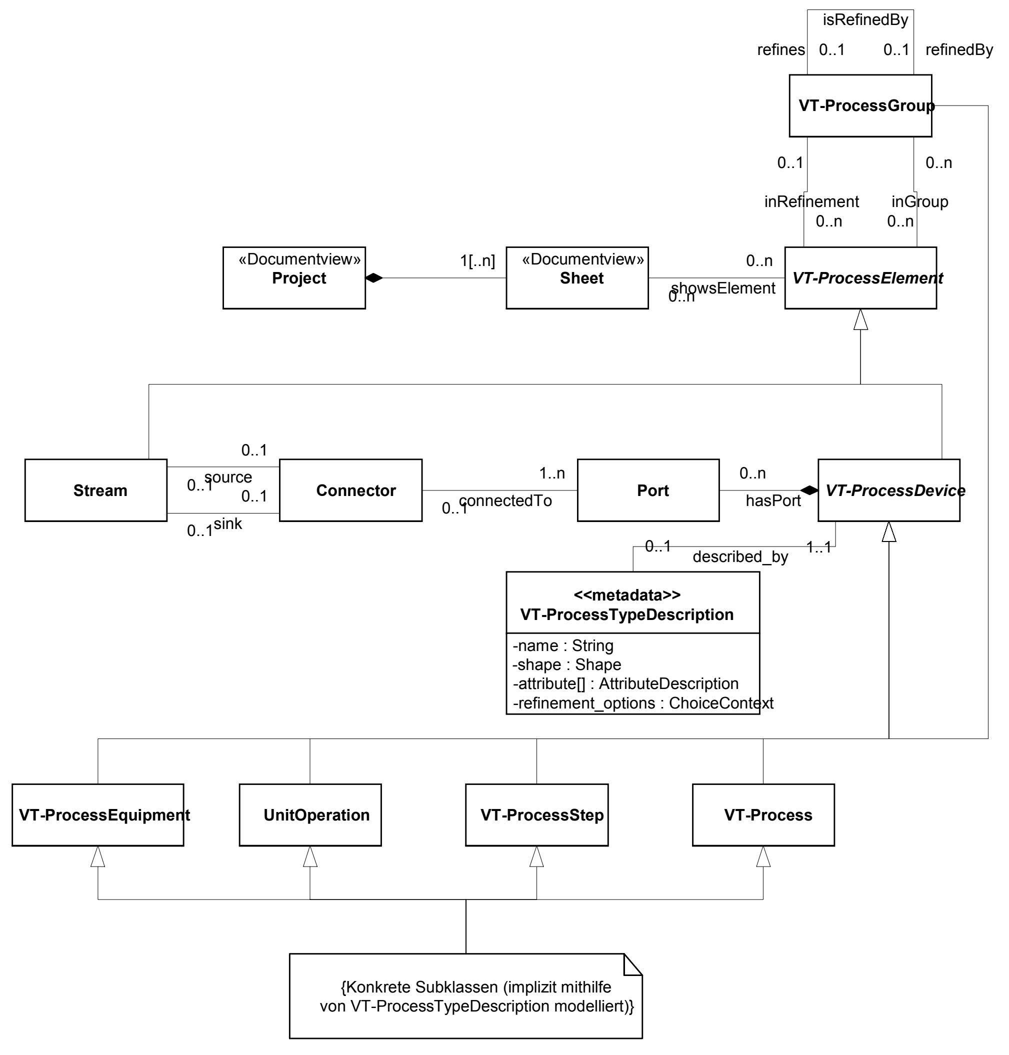


***Abb. 72**: Datenmodell des Fließbildeditors*

Prozesselemente

Abb. 72 zeigt das Fließbild-Datenmodell. Ein Verfahrens- bzw. Anlagenkonzept ist durch eine Menge von miteinander in Beziehung stehenden VT-Prozesselementen (`VT-ProcessElement`) charakterisiert. Die abstrakte Oberklasse `VT-ProcessElement` zerfällt in zwei wesentliche Subklassen: `VT-ProcessDevices`, die die Hauptfunktionalität eines Verfahrens repräsentieren, und `Streams`, die für den Stoff-, Energie- oder Materialtransport zwischen `VT-ProcessDevices` sorgen. Instanzen der Subklassen von `VT-ProcessDevice` werden im Fließbildgraphen als Knoten und `Stream`-Instanzen als Verbindungslinien zwischen den Knoten dargestellt.

Im Laufe des Modellierungsprozesses werden der Menge von `VT-ProcessElement`-Instanzen zwei wesentliche Strukturen aufgeprägt, die vom Fließbildwerkzeug zu verwalten sind: die Verknüpfungsstruktur und die Verfeinerungsstruktur.

Verknüpfungsstruktur

Die Verknüpfungsstruktur gibt an, wie `VT-ProcessDevices` über `Streams` miteinander verknüpft sind. Jeder `Stream` besitzt zwei Konnektoren (`Connector`), von denen einer die Quelle (source) und der andere die Senke (sink) des `Streams` definiert. Dadurch ist die (Fluss-)Richtung eines `Streams` bestimmt. Die Verknüpfungsstellen eines `VT-ProcessDevice` werden als `Port` bezeichnet. Ein `VT-Process-`

Device hat mindestens einen Port. Jeder Port ist dadurch näher charakterisiert, ob er als Eingangsport oder Ausgangsport (Attribut direction) fungiert und ob er zwingend erforderlich oder optional ist. Die Anzahl der Ports sowie deren Eigenschaften hängen von der spezifischen VT-ProcessDevice-Subklasse ab. Die Verknüpfung zwischen einem Stream und einem VT-ProcessDevice erfolgt, indem der Connector eines Streams an den Port eines VT-ProcessDevice gebunden wird. Hierbei ist darauf zu achten, dass ein Source-Connector nur mit einem Ausgangsport und ein Sink-Connector nur mit einem Eingangsport verbunden wird. Jeder Connector kann an mehrere Ports geknüpft werden und umgekehrt. Das hängt mit der nachfolgend beschriebenen Verfeinerung von VT-ProcessDevices und Streams zusammen.

Verfeinerungsstruktur

Die Verfeinerungsstruktur beschreibt die Detaillierung von VT-ProcessDevices. und Streams. Drei wesentliche Arten von Detaillierungsbeziehungen sind zu unterscheiden: Dekomposion, Anreicherung, Konkretisierung und Realisierung.

- **Dekomposition**: Dekomposition tritt bei der Verfeinerung eines Verfahrenskonzepts auf der Ebene des Grundfließbilds auf. Ein VT-Process wird dekomponiert in eine Menge von VT-ProcessSteps und/oder UnitOperations sowie Streams. Ebenso können VT-ProcessSteps und Streams dekomponiert werden. UnitOperations sind atomare funktionale Einheiten, die nicht weiter dekomponiert werden können. Der Gesamtprozess (Instanz der Klasse VT-Process) ist stets die Wurzel einer Dekompositionshierarchie, kann also selbst nicht als Teil einer Dekomposition auftauchen.
- **Anreicherung**: Bei der Anreicherung bleibt eine Gruppe von VT-Prozesselementen auch auf der nächsten Verfeinerungsstufe vollständig erhalten, wird jedoch um zusätzliche VT-Prozesselemente angereichert.
- **Konkretisierung**: Bei der Konkretisierung wird ein VT-ProcessStep durch einen spezielleren VT-ProcessStep oder UnitOperation beschrieben, der die gleiche Funktion erfüllt, dessen Prozessverhalten jedoch konkretisiert wird und der „näher" an der apparativen Umsetzung ist.
- **Realisierung**: Die Realisierungsbeziehung beschreibt die apparatetechnische Umsetzung von VT-ProcessSteps und/oder UnitOperations mit Hilfe einer (Gruppe von) Anlagen.

Abb. 71 auf Seite 253 liefert Beispiele für alle vier Verfeinerungsarten.

Zur Beschreibung von Verfeinerungen führen wir das Konzept der VT-Prozessgruppen (VT-ProcessGroup) ein, zwischen denen die oben beschriebenen Verfeinerungsbeziehungen eingerichtet werden können (mithilfe der Assoziationsklasse isRefinedBy und den entsprechenden Subklassen). Die Zugehörigkeit eines VT-Prozesselements zu einer VT-Prozessgruppe wird durch die Assoziationen inRefinement und inGroup repräsentiert. Der Unterschied zwischen diesen Assoziationen liegt in den jeweiligen Rollen, die die betreffenden VT-Prozessgruppen innerhalb der Verfeinerungshierarchie spielen: inRefinement bedeutet, dass das VT-Prozesselement zu einer *verfeinernden* VT-Prozessgruppe gehört. Die Assoziation inGroup drückt aus, dass das VT-Prozesselement einer *verfeinerten* VT-Prozessgruppe angehört. Ein VT-Prozesselement kann zu beliebig vielen verfeinerten VT-Prozessgruppen gehören, aber höchstens zu einer Gruppe, die selbst

wieder eine Verfeinerung einer anderen darstellt [49]. Die Klasse VT-ProcessGroup ist eine Subklasse von VT-ProcessDevice. Daher hat eine komplexe Gruppe selbst wieder eine Menge von Ports, die an Konnektoren von Strömen gebunden werden können.

Erweiterbares Typsystem

Als wichtige Voraussetzung für ein erweiterbares Typsystem wurden im Datenmodell des Fließbildeditors nur die obersten Klassen des Bausteintypsystems explizit festgelegt. Auf der Ebene der Grundfließbilder (funktionale Sicht) sind dies die Klassen VT-Process, VT-ProcessStep und UnitOperation. Auf Fließbildebene wurde lediglich die Klasse VT-ProcessEquipment als abstrakte Oberklasse für konkrete Apparate und Maschinentypen definiert. Eine explizite Abbildung konkreter Bausteintypen wie z.B. Reaktion, Trennung, Destillationskolonne etc. hätte zu einem sehr unübersichtlichen Schema geführt. Außerdem hätte die dynamische Erweiterung um neue Bausteintypen jeweils eine Schemaänderung nach sich gezogen.

Metamodellierung der Bausteintypen

Konkrete Bausteintypen werden stattdessen *implizit* mithilfe der Klasse VT-ProcessDeviceTypeInfo, definiert, deren Instanzen die aktuell verfügbaren Bausteintypen beschreiben. Die Typbeschreibungen sind dem Fließbildwerkzeug zur Laufzeit zugänglich; durch Instanziierung können auf einfache Weise neue Bausteintypen definiert werden. Dazu müssen eine Reihe von Typbeschreibungsmerkmalen angegeben werden, z.B. die Kategorie des Bausteintyps (VT-ProcessStep, UnitOperation, VT-ProcessEquipment), den Namen, das Darstellungselement und eine Liste von Attributbeschreibungen. In der aktuellen Implementierung wurden lediglich getypte Name-Wert-Paare zur Attributierung von VT-ProcessDevice-Subklassen realisiert. Ein sehr viel weitergehendes Konzept zur dynamischen Attributverwaltung wurde in [Baum00] entwickelt, jedoch aus Aufwandsgründen nicht innerhalb der PRIME-IMPROVE-Umgebung verwirklicht wurde.

Hinsichtlich der Verfeinerungsverträglichkeit zwischen Bausteinen bzw. Bausteingruppen sind nur elementare Konsistenzbedingungen im Fließbild-Datenmodell hinterlegt, z.B. dass eine UnitOperation als atomarer Funktionsbaustein nicht weiter *dekomponiert*, sondern nur durch Instanzen von VT-ProcessEquipment *realisiert* werden kann. Weitergehende Unterstützung erhält der Verfahrensingenieur durch M-Prozessfragmente, die die systematische Generierung von Verfahrensalternativen und deren Analyse und Simulation anleiten. Das Wissen, welche Verfeinerungs-Prozessfragmente anwendbar sind, ist nicht im Fließbild-Datenmodell selbst verankert, sondern wird in einem so genannten Process Data Warehouse (PDW) verwaltet [WeBa99]. Das Process Data Warehouse aggregiert Informationen aus dem Fließbildmodell mit Daten aus anderen Werkzeugen (Simulatoren, Stoffdatenbanken) und realisiert so eine erweiterte, zweistufige Situationsanalysefunktionalität, die die Situationserkennung im PRIME-Framework komplementiert. Weitergehende Informationen zum Process Data Warehouse und seiner Anbindung an das Fließbild-Datenmodell sind in [JaLW00] zu finden.

---

[49] Implizit bedeutet dies, dass zu einer Prozessgruppe eine beliebige Anzahl alternativer Verfeinerungen existieren können, während zu jeder Prozessgruppe höchstens eine übergeordnete Gruppe existieren kann. Wir erhalten dadurch einen Baum von alternativen Verfeinerungen.

Umgang mit dem Fließbildmodell aus Benutzersicht

Die bisher beschriebenen Klassen definieren den *Inhalt* eines Fließbildmodells, das über beliebig viele Verfeinerungsebenen strukturiert sein kann. Im Fließbildwerkzeug kann der Benutzer zu einem Zeitpunkt jeweils nur einen bestimmten Ausschnitt dieses Modells betrachten, bei dem die hierarchische Struktur des Modells auf ein „flaches" Zeichenblatt herunter gebrochen wird. Der Benutzer verändert den aktuell betrachteten Ausschnitt eines Fließbildmodells, indem er neue VT-Prozesselemente und -gruppen einfügt und zwischen unterschiedlichen Verfeinerungsebenen hin- und her navigiert. Eine Besonderheit des Fließbildeditors besteht darin, dass das Ein- und Ausblenden einer verfeinerten VT-Prozessgruppe innerhalb des gleichen Fensters möglich ist. Dadurch bleibt die Umgebung des betrachteten Fließbildausschnitts in der Ansicht erhalten.

Dokumentenorientierte Erweiterung

Ein Fließbild definiert lediglich eine temporäre Sicht im Fließbildeditor, die der Benutzer durch Einfüge- und Navigationsoperationen innerhalb seines Fließbildmodell hergestellt hat. Aus administrativer Sicht ist es jedoch manchmal erforderlich, einen bestimmten Darstellungszustand des Fließbildmodells als festes Fließbild-*Dokument* einzufrieren. Aus diesem Grund haben wir das zusätzliche Konzept eines *Zeichenblatts* (`Sheet`) eingeführt, mit dessen Hilfe dem Fließbildmodell eine dokumentenorientierte Sicht aufgeprägt werden kann. Ein Zeichenblatt wird zu einem bestimmten Zeitpunkt vom Benutzer angelegt und aggregiert alle aktuell sichtbaren VT-Prozesselemente (`VT-ProcessDevices`, `Streams`, `VT-ProcessGroups`) zusammen mit ihren Layout-Informationen. Ein `Sheet` kann mit einem Bezeichner benannt werden und jederzeit wieder hergestellt werden. Mehrere `Sheets` können zu einem *Projekt* (`Project`) zusammengefasst werden.

8.2.3 Positionierung von PRIME-IMPROVE im SFB-Prototypen

Die PRIME-IMPROVE-Umgebung ist Teil eines integrierten Prototypen, der zusammen mit anderen Teilprojekten des Sonderforschungsforschungsbereich IMPROVE entwickelt wurde und auf der ersten Begehung des SFBs sowie auf mehreren Workshops mit Industriebeteiligung erfolgreich demonstriert wurde. Abb. 73 gibt einen Überblick über die Grobarchitektur der PRIME-IMPROVE-Umgebung. Über die Integration der vollständig und partiell prozessintegrierbaren Werkzeuge hinaus wurden eine Reihe von Schnittstellen zu den anderen Komponenten der SFB-Gesamtumgebung realisiert, mit deren Hilfe die Unterstützungsmechanismen der unterschiedlichen SFB-Teilprojekte synergetisch verzahnt werden konnten. Beispielsweise bietet die Prozessmaschine eine CORBA-Schnittstelle an, über die das Frontend des arbeitsplatzübergreifenden Projekt-Administrationssystems die Ausführung von M-Prozessfragmenten in der PRIME-IMPROVE-Umgebung anstoßen und überwachen kann. Die Funktionalität dieser Schnittstellen orientiert sich an dem entsprechenden Standard der WfMC.

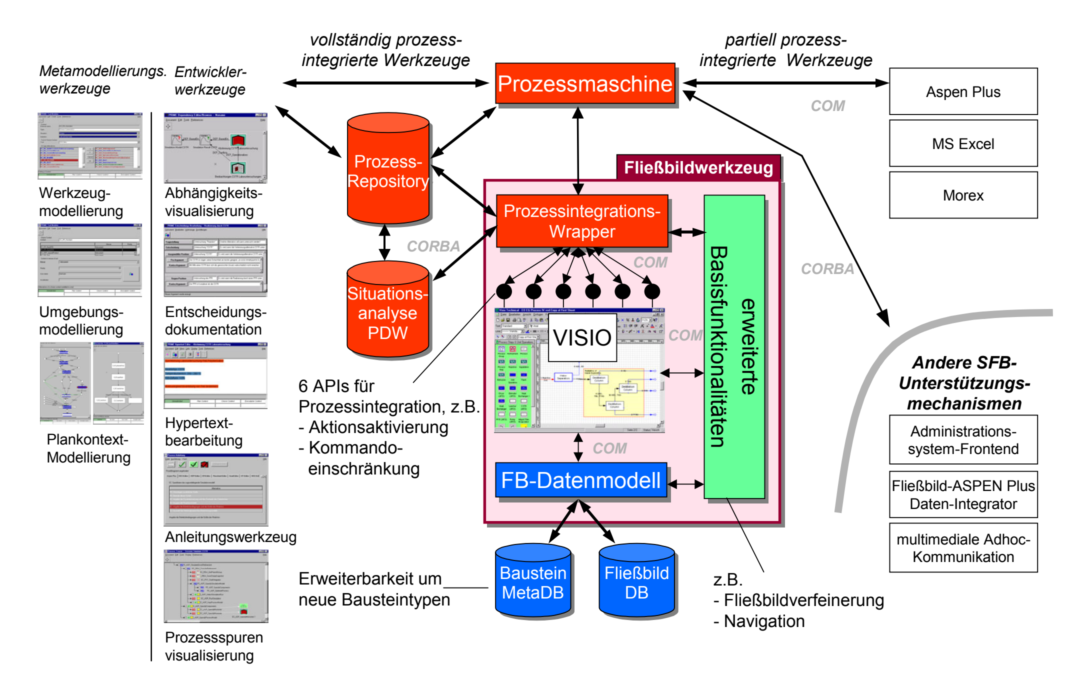


***Abb. 73**: Gesamtarchitektur der PRIME-IMPROVE-Umgebung*

8.3 Beispielsitzung

Wir beschreiben nun eine kurze Beispielsitzung, um die Funktionalität der PRIME-IMPROVE-Umgebung und die Vorteile der Prozessintegration aus Benutzersicht zu demonstrieren. Das betrachtete Szenario besteht aus zwei Teilen. Der erste Teil behandelt die *Definition* eines neuen M-Prozessfragments und dessen Integration in den Fließbildeditor über das Umgebungsmodell. Der zweite Teil illustriert die methodische Anleitung des Benutzers während der *Ausführung* des vorher definierten M-Prozessfragments.

8.3.1 Definition eines Prozessfragments

8.3.1.1 Ziel der Prozessunterstützung

Zu den Basisfunktionalitäten des Fließbildeditors gehört das Anlegen einer Verfeinerungsgruppe für einen VT-Prozessbaustein. Diese Funktionalität wird durch den Ausführungskontext `RefineProcessDevice` realisiert, dessen Situation gültig ist, sobald eine Instanz der Klasse `VT-ProcessDevice` vom Benutzer ausgewählt wurde. Die Funktionalität dieses Ausführungskontexts besteht lediglich darin, eine neue, leere Verfeinerungsgruppe für den ausgewählten VT-Prozessbaustein zu kreieren und in der grafischen Ansicht auf die Ebene der neu angelegten Gruppe zu navigieren. Eine darüber hinaus gehende Ablaufunterstützung liefert der Kontext nicht.

Der Methodeningenieur erkennt, dass in der vorliegenden Konfiguration des Fließbildwerkzeugs häufig komplexe Verfeinerungshierarchien mit vielen alternativen Verfeinerungen für einen VT-Prozessbaustein entstehen, wobei sich jedoch die einzelnen Alternativen häufig nur geringfügig unterscheiden und die Gründe für die Auswahl einer Alternative schlecht dokumentiert sind.

Um das Anlegen neuer Verfeinerungsalternativen konsistent und nachvollziehbar zu machen, ersetzt der Methodeningenieur den Ausführungskontext RefineProcessDevice durch einen neuen Plankontext RefineProcessDeviceWithDecision. Hinter diesem Plankontext steht die Überlegung, dass die M-Prozessunterstützung den Verfahrensingenieur bei der Verfeinerung eines VT-Prozessbausteins zunächst auf eventuell schon existierende Verfeinerungen hinweisen sollte. Weiterhin sollte die M-Prozessunterstützung den Verfahrensingenieur mit allen Hintergrundinformationen (Entscheidungen, Argumente, informelle Anforderungen, Simulationsergebnisse etc.) versorgen, mit denen bereits existierende Verfeinerungen des betreffenden VT-Prozessschritts dokumentiert wurden. Eine Analyse dieser Informationen kann dazu führen, dass der Verfahrensingenieur auf das Anlegen einer weiteren Verfeinerung verzichtet. Wird dennoch eine Verfeinerung angelegt, dann müssen die Argumente, die für oder gegen eine weitere Verfeinerungsalternative sprechen, modifiziert und ergänzt sowie die damit verbundenen Änderungen protokolliert werden.

8.3.1.2 Prozessmodellierung

Plankontext RefineProcessDeviceWithDecision

Die zuvor beschriebene M-Prozessanleitung wird durch den in Abb. 74 dargestellten Plankontext RefineProcessDeviceWithDecision realisiert. Für die Modellierung dieses Plankontexts verwendet der Methodeningenieur SLANG-Netze. Dieser Formalismus eignet sich gut zur Modellierung der in dem Ablauf auftretenden mehrstufigen Datenabhängigkeiten.

Notation

Transitionen, die an eine Kontextkomponente gebunden sind, sind dadurch gekennzeichnet, dass der Name des Kontextes als Transitionsnamen angegeben ist, wohingegen ungebundene Transitionen, die z.B. für die Modellierung von Verzweigungen erforderlich sind, namenlos sind. Eine rot dargestellte Transition verfügt über einen Wächter, mit dem logische Bedingungen an die Stellen im Vorbereich gestellt werden. Der Vor- und Nachbereich einer gebundenen Transition entsprechen der Ein- bzw. Ausgangsschnittstelle, wobei an der Kante zwischen der Stelle und der Transition jeweils der Rollenname angegeben ist. Stellen, die nicht an einen Situationsteil gebunden sind, sind dadurch gekennzeichnet, dass der Rollenname in <<Klammern>> steht, wohingegen der Rollenname einer Stelle, die als Situationsteil einer Stellvertreterschnittstelle gebunden ist, unverändert dargestellt wird.

Aktivierungsbedingung

Der Plankontext RefineProcessDeviceWithDecision wird aktiviert, wenn im Fließbildwerkzeug das Projekt (Project) mit einem Arbeitsblatt (Sheet) geöffnet ist, das entsprechende VT-Prozesselement (Device) des Arbeitsblattes selektiert ist (Situation) und der Verfahrensingenieur M-Prozessunterstützung für die Verfeinerung der VT-Prozessgruppe anfordert (Intention Refine).

Das aktuelle Project, Sheet und Device bilden den Vorbereich der Starttransition, die das SLANG-Aktivitätsnetz startet. Mit dem Schalten der Transition werden die aktuellen Marken dem Vorbereich des Ausführungskontextes EC_FBW_CreateGroup zugewiesen. Dieser Ausführungskontext bestimmt für ein selektiertes Gerät die zugehörige VT-Prozessgruppe oder legt eine neue an, sofern diese noch nicht existiert. Im nächsten Schritt werden für diese VT-Prozessgruppe mit dem Ausführungskontext EC_FBW_GetRefinementsOfGroup alle Verfeinerungen

bestimmt, die als Alternativen für die Realisierungen der VT-Prozessgruppe bereits vorhanden sind.

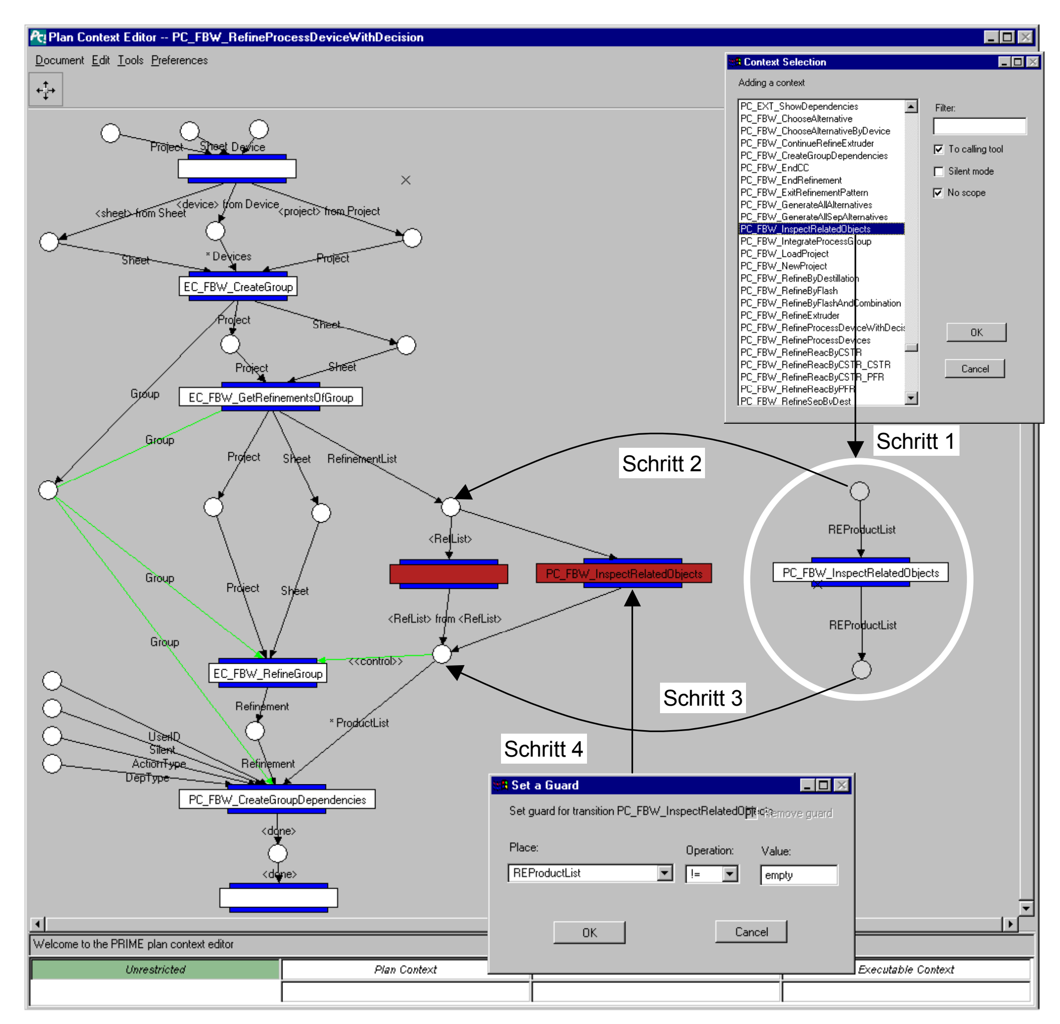


Abb. 74: SLANG-Modell des Plankontexts PC_FBW_RefineProcessDeviceWithDecision

Im nächsten Schritt sieht der Plankontext eine bedingte Verzweigung vor, die vom Ergebnis des Kontextes `EC_FBW_GetRefinementsOfGroup` abhängt und durch Wächterbedingungen in den adjazenten Transitionen realisiert wird. Die Liste der Verfeinerungen `RefinementList` kann zum einen leer sein, falls für die VT-Prozessgruppe noch keine Verfeinerung angelegt wurde, oder beinhaltet mindestens eine Verfeinerung für die ursprünglich ausgewählte VT-Prozessgruppe. Falls keine Verfeinerung existiert, dann kann die VT-Prozessgruppe ohne Rücksicht auf vorhandene Entscheidungen verfeinert werden. Dieser Fall wird durch die linke der beiden rot dargestellten Transitionen in Abb. 74 abgedeckt. Diese Transition dient ausschließlich der Kontrollflusssteuerung und ist nicht an eine Kontextkomponente gebunden.

Falls jedoch bereits eine Verfeinerung existiert, soll der Verfahrensingenieur zunächst alle mit dieser Verfeinerung in Zusammenhang stehenden Zusatzinformationen betrachten. Dieser Vorgang wird durch den Plankontext `PC_FBW_InspectRelatedObjects` angeleitet (rechte der beiden roten Transitionen), welcher zu einer gegebenen Menge von Produkten (in diesem Fall die zu verfeinernde VT-Prozessgruppe) alle abhängigen Produkte ermittelt und dem Benutzer zur Ansicht und Bearbeitung anbietet.

In beiden Fällen wird, sofern der Plankontext PC_FBW_InspectRelatedObjects nicht vom Benutzer abgebrochen wurde, mit dem Kontext EC_FBW_RefineGroup eine Verfeinerung der ursprünglichen VT-Prozessgruppe angelegt. Nachdem die Verfeinerung im Fließbildwerkzeug angelegt wurde, werden Abhängigkeiten zwischen der ursprünglichen VT-Prozessgruppe, der neu angelegten VT-Prozessgruppe und den ggf. im Plankontext PC_FBW_InspectRelatedObjects modifizierten Zusatzinformationen (Hypertextdokumente, Entscheidungen) angelegt. Die Aktualisierung der Abhängigkeitsstruktur erfolgt über den Plankontext PC_FBW_CreateGroupDependencies. Bei der Aktualisierung sind weiterhin die Benutzerkennung, der Aktivierungsmodus des Werkzeuges sowie der Beziehungstyp der Produkte anzugeben, die als vorinitialisierten Marken vorliegen.

Mit der Termination dieses Plankontextes wird die modellierte M-Prozessunterstützung beendet.

Komponentenbasierte Einbettung existierender Prozessfragmente

Obwohl das beschriebene SLANG-Netz einen komplexen Ablauf modelliert und mehrere Werkzeuge involviert sind, ist der dafür erforderliche Modellierungsaufwand relativ gering und das resultierende M-Prozessmodell vergleichsweise übersichtlich. Eine wesentlicher Grund dafür liegt in der sprachübergreifenden Wiederverwendung bereits existierender M-Prozessfragmente. In dem dargestellten Szenario greift der Methodeningenieur beispielsweise auf die Plankontextkomponenten PC_FBW_CreateDependencies und PC_FBW_InspectRelatedObjects zurück, die in den Sprachen C++ bzw. UML-Statecharts realisiert wurden.

Einbettung einer Kontextkomponente

Am Beispiel des Plankontexts PC_FBW_InspectRelatedObjects wollen wir die Einbettung einer Kontextkomponente bei der Plankontextdefinition genauer betrachten (rechter Teil von Abb. 74). Nach der Auswahl der Komponente aus einer sprachneutralen Komponentensammlung wird vom Modellierungswerkzeug eine Stellvertreterschnittstelle erzeugt, die die Komponente mit den Konzepten der Verwendungssprache SLANG darstellt (Schritt 1). Daraufhin müssen die Situationsteile an den Datenfluss im Plankontext angeschlossen werden. Dazu verbindet der Methodeningenieur die Situationsteile in den Rollen REProduct und REProductList mit existierenden Stellen im SLANG-Plankontext (Schritt 2 und 3). Dabei muss der Typ der verbundenen Stelle mit dem Typ des Situationsteils übereinstimmen. Die abgebildete Stellvertreterkomponente, deren Ein- und Ausgangsschnittstelle sowie die Situationsteile sind über Sprachbindungen an die sprachneutrale Schnittstellenbeschreibung gebunden. Im abschließenden Schritt 4 gibt der Methodeningenieur eine zusätzliche Situationsbedingung in Form eines Transitionswächters an.

Plankontext InspectRelatedObjects

Der Plankontext PC_FBW_InspectRelatedObjects selbst wurde nicht als SLANG-Netz, sondern als UML-Statechart modelliert, da seine Struktur nicht von komplexen, mehrstufigen Datenabhängigkeiten, sondern von einfachen Steuerbedingungen geprägt ist.

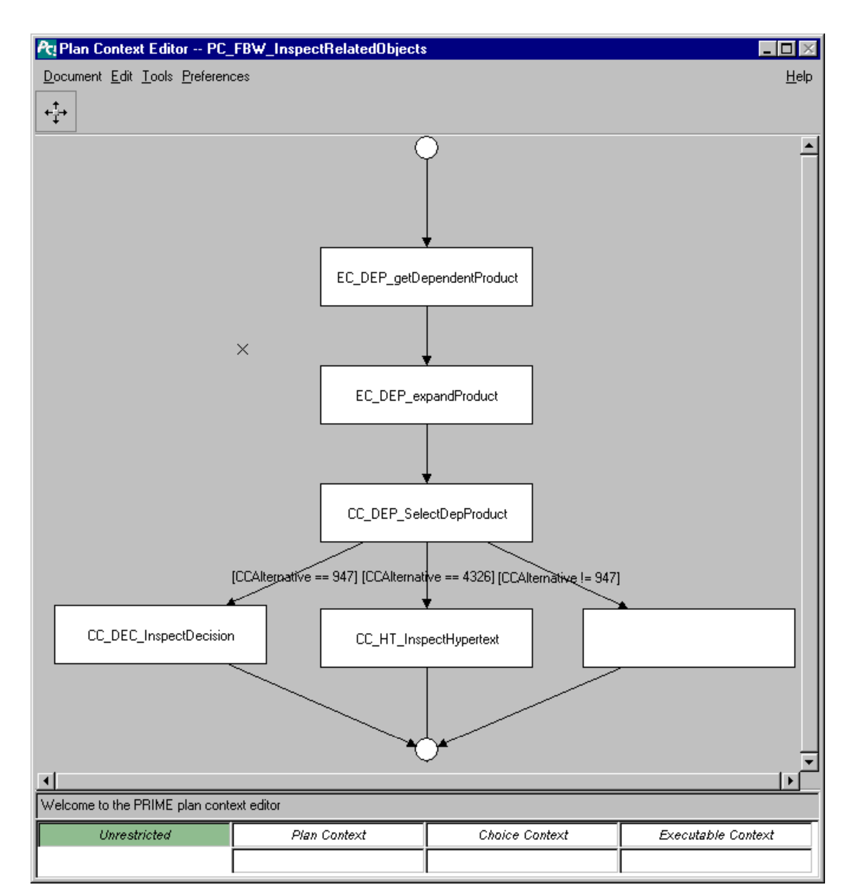


***Abb. 75**: UML-Statechart-Modell des Plankontexts InspectRelatedObjects*

Abb. 75 zeigt den Plankontext PC_FBW_InspectRelatedObjects als UML-Zustandsdiagramm, dessen Eingangsschnittstelle eine Instanz des generischen Typen Product erwartet und dessen Ausgangsschnittstelle eine Produktliste ProductList liefert. Um das gewünschte Verhalten des Plankontextes zu realisieren, müssen zunächst diejenigen Produkte ermittelt werden, die mit dem Eingangsprodukt in einer Abhängigkeitsbeziehung stehen (EC_DEP_GetDependentProducts). Die Liste der abhängigen Produkte wird mit der nachfolgenden Werkzeugaktion EC_DEP_ExpandProduct im Abhängigkeitseditor zusammen mit den Beziehungen zu der Verfeinerung angezeigt. Daraufhin wird der Abhängigkeitseditor angewiesen, in den Entscheidungskontext CC_DEP_Explore überzugehen, so dass der Benutzer die angezeigten Produkte mit dem Abhängigkeitseditor untersuchen kann. Dieser Entscheidungskontext besteht aus Kontextalternativen, mit dem der Benutzer die Produkte und deren Abhängigkeiten untersuchen kann (hier nicht im Detail dargestellt). Der Benutzer kann frei zwischen allen Alternativen wählen und solange im Entscheidungskontext CC_DEP_Explore arbeiten, bis er als Kontextalternative entweder den Kontext CC_DEC_InspectDecision oder CC_HT_InspectHypertext wählt[50] und damit die wiederholte Aktivierung des Entscheidungskontextes beendet. Mit diesen Entscheidungskontexten kann sich der Benutzer Entscheidungs- bzw. Hypertextdokumente in den jeweiligen Editoren ansehen und bei Bedarf modifizieren. Beispielsweise beinhaltet der Entscheidungskontext CC_InspectDecision den Plankontext PC_ReviseDecision, mit dem die Revision einer früheren Entscheidung dokumentiert werden kann.

[50] Diese beiden alternativen Kontexte haben die gleiche Intention (Inspect), aber unterschiedliche Situationen (DecisionSelected bzw. HypertextSelected)

8.3.1.3 Werkzeugmodellierung

Damit der neu definierte Plankontext PC_RefineProcessDeviceWithDecision überhaupt aus dem Fließbildwerkzeug heraus aktiviert werden kann, muss er über das Umgebungsmodell mit dem Werkzeugmodell des Fließbildwerkzeugs in Beziehung gesetzt werden.

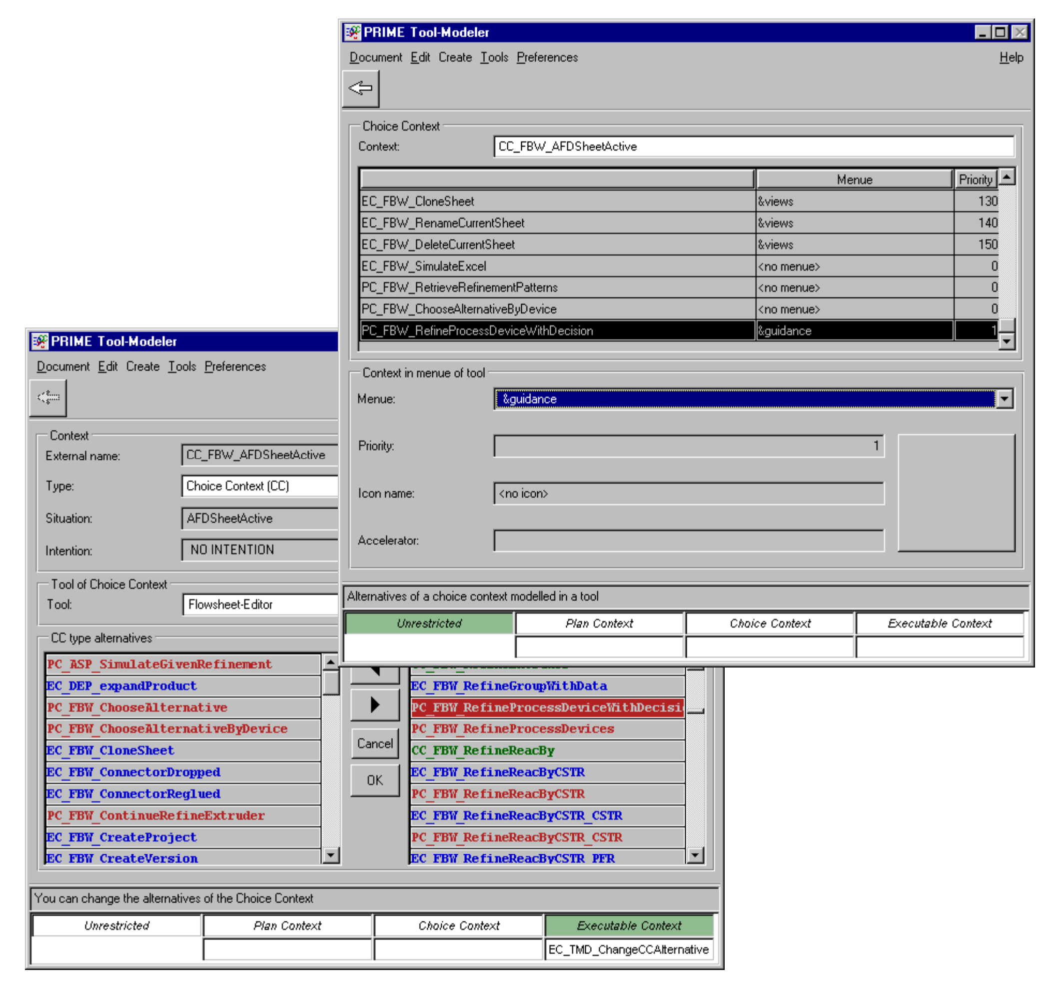


***Abb. 76**:*
Der neue Plankontext wird als Alternative zum Standardkontext des Fließbildwerkzeugs hinzugefügt

Zunächst wird der neue Plankontext als Alternative zum Entscheidungskontexts CC_FBW_AFDSheetActive (Abb. 76, unten links). Dies ist ein Standardkontext des Fließbildwerkzeugs, der immer dann aktiv ist, wenn aktuell ein Grundfließbild geladen ist. In einem zweiten Schritt wird festgelegt, dass die Intention des neuen Plankontexts im Menü „Guidance/Prozessanleitung“ als Menüpunkt erscheinen soll (Abb. 76, oben rechts).

8.3.2 Ausführung eines Prozessfragments

Nachdem wir im vorangegangenen Abschnitt die Modellierung des M-Prozessfragments RefineProcessDeviceWithDecision aus Sicht des Methodeningenieurs betrachtet haben, schlüpfen wir nun in die Rolle des Verfahrensingenieurs, der durch das M-Prozessfragment bei der Verfeinerung eines VT-Prozessbausteins angeleitet wird.

Das Beispiel betrachtet den Entwurf des Aufbereitungsteils (Compounding) einer Anlage zur Herstellung von Nylon. Abb. 77 zeigt den Fließbildeditor, in dem

der VT-Prozessschritt `Compounding` in einer ersten Verfeinerung zu sehen ist. Der Eingangsstrom des Aufbereitungsschritts stammt aus dem vorangeschalteten Reaktionsschritt und führt das gewünschte Endprodukt Nylon sowie nicht umgesetzte Restanteile der Edukte Wasser und Capro-Lactam (CL) mit sich. Innerhalb des Aufbereitungsschritts wird zunächst ein Großteil der Edukte Wasser und Capro-Lactam mit Hilfe eines `Wiped Film Evaporator` abgetrennt und dann das verbleibende, nur noch leicht durch CL verunreinigte Nylon in einem `Extruder` unter Beigabe von Zusatzsatzstoffen weiterverarbeitet. Das im Wiped Film Evaporator abgetrennte Wasser-CL-Gemisch (Strom 21) wird durch einen weiteren Trennschritt (`Separation`) aufgespaltet, um den teuren Ausgangsstoff CL wieder der Reaktion zurückzuführen, während das verbleibende Wasser als Abfallstoff abgeführt wird.

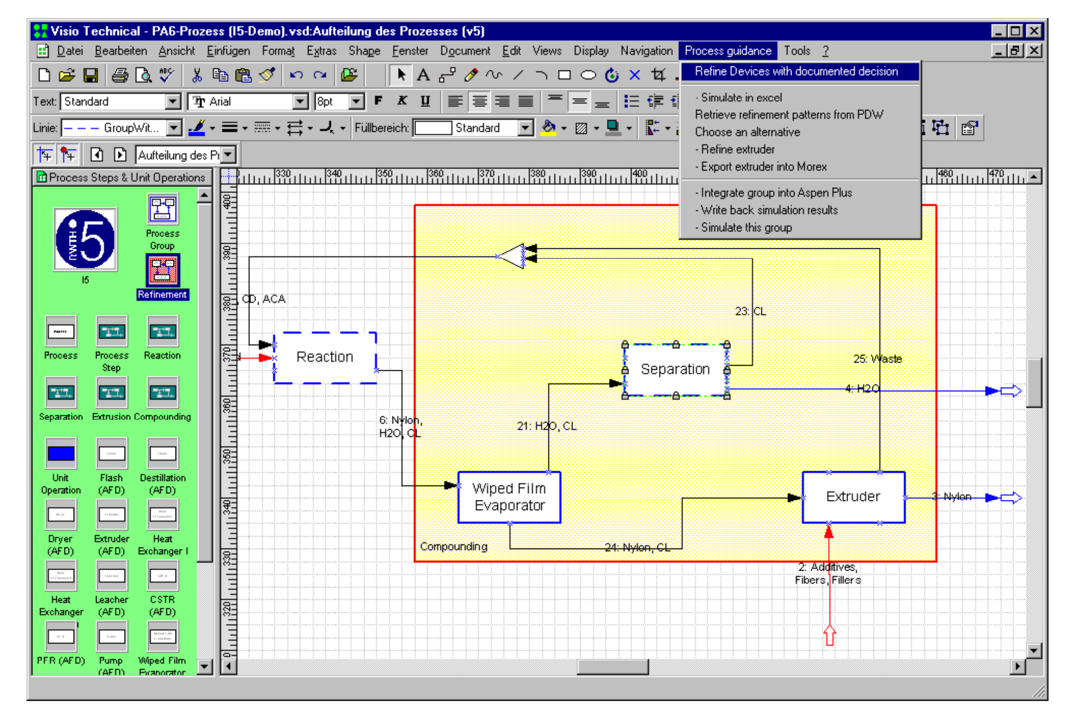


Abb. 77: Aktivierung des Plankontexts PC_FBW_RefineProcessDeviceWithDecision im Fließbildeditor

In dem betrachteten Beispielszenario hat der Verfahrensingenieur die Aufgabe, den VT-Prozessschritt `Separation` durch eine apparatetechnische Realisierung weiter zu verfeinern. Dazu selektiert er den VT-Prozessschritt und ruft im Menü `Prozessanleitung` das Kommando `Verfeinere mit dokumentierter Entscheidung` auf.

Diese Auswahl entspricht dem oben definierten Plankontext `RefineProcessDeviceWithDecision`, der nun im Fließbildwerkzeug verfügbar. Der `ContextMatcher` des Fließbildwerkzeugs (bzw. des Prozessintegrations-Wrappers) gleicht die Auswahl mit den Kontextdefinitionen im Umgebungsmodell ab und aktiviert den Plankontext. Die Prozessmaschine allokiert daraufhin die benötigten Werkzeuge und startet die Ausführung des Plankontexts `RefineProcessDeviceWithDecision`. Da dieses M-Prozessfragment in SLANG modelliert ist, wird in der Sprachelemente-Schicht des Prozessmaschinen-Rahmenwerks das entsprechende SLANG-Netz geladen und instanziiert.

Die ersten beiden Schritte des SLANG-Netzes entsprechen Ausführungskontexten, die von der Prozessmaschine automatisch im Fließbildwerkzeug aktiviert werden. Dabei wird für den ausgewählten VT-Prozessschritt `Separation` eine VT-

Prozessgruppe kreiert bzw. eine eventuell schon vorhandene VT-Prozessgruppe ermittelt (EC_FBW_CreateProcessGroup) und alle bereits existierenden Verfeinerungen dieser Gruppe angefragt (EC_FBW_GetRefinementsOfGroup.). Im vorliegenden Fall gab es bereits eine Verfeinerung für Separation, so dass der Ausführungskontext EC_FBW_GetRefinementsOfGroup eine nichtleere Liste von Verfeinerungen (RefinementList) zurückliefert. Diese enthält eine Verfeinerungsgruppe mit dem Namen „Realization of Separation by Destillation"

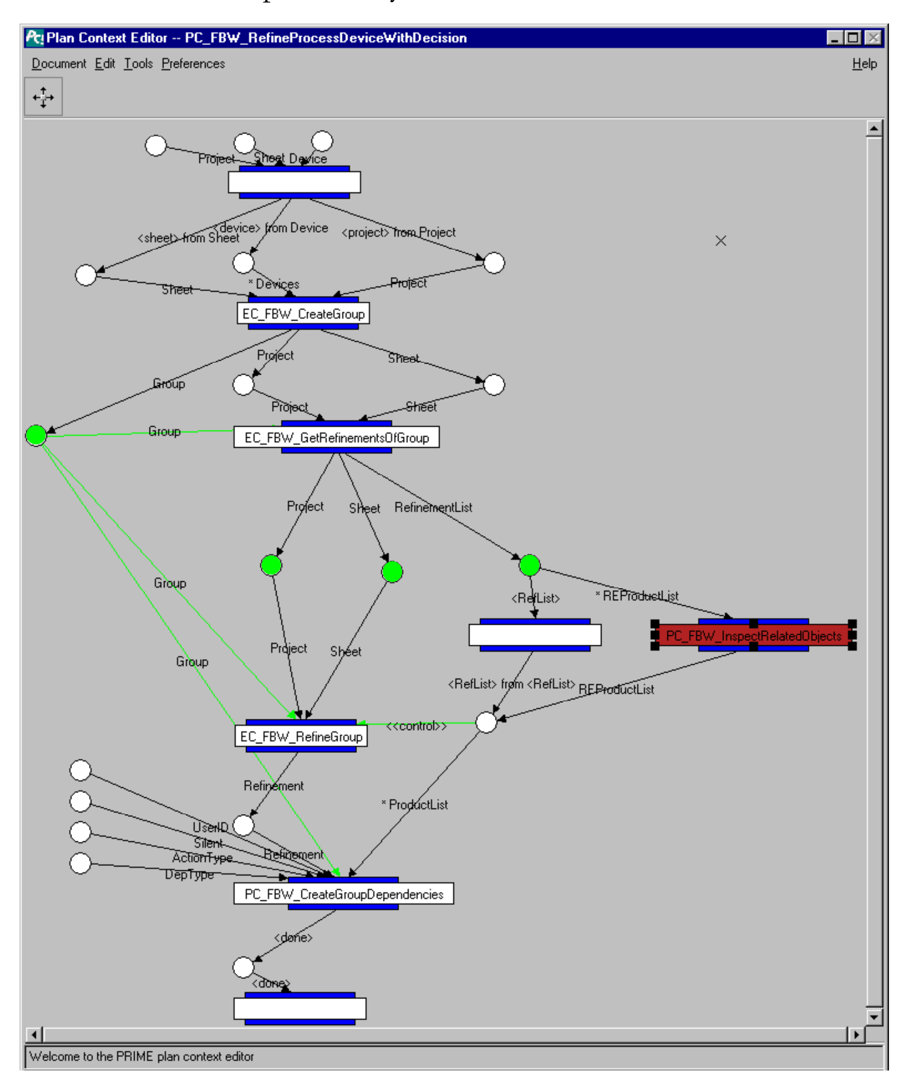


***Abb. 78**: Ausführung des Plankontexts Refine-ProcessDeviceWith-Decision*

Abb. 78 zeigt den Zustand der Plankontextinterpretation nach diesen ersten beiden Schritten[51]. Gemäß der aktuellen Belegung der Stellen und der Definition der Wächterbedingungen kann aktuell nur die Transition, die den eingebetteten Plankontext PC_FBW_InspectRelatedObjects repräsentiert, schalten. Wie oben beschrieben, wurde dieser Plankontext mithilfe von UML-Statecharts modelliert, so dass das Prozessmaschinen-Rahmenwerk das UML-Startchart instanziiert und die

[51] Die Plankontext-Editoren sind in der Lage, den Ausführungszustand eines Prozessfragments dynamisch zu visualisieren. Dazu müssen sie sich zuvor beim Prozessspurenserver registriert haben, der im Hintergrund die verteilte Ausführung von Prozessfragmenten protokolliert und Ereignisse wie den Start und das Beenden eines Kontexts an alle registrierten Interessenten weiterleitet.

Kontrolle an die entsprechenden UML-Interpreterobjekte der sprachspezifischen Schicht übergibt.

Innerhalb des Plankontexts PC_FBW_InspectRelatedObjects werden alle Entwurfsobjekte ermittelt, die zu der gefundenen Separation-Verfeinerung in Beziehung stehen, und angezeigt. Hierzu werden die Ausführungskontexte `EC_DEP_getDependentProduct` bzw. `EC_DEP_expandProduct` im Abhängigkeitseditor von der Prozessmaschine aktiviert, so dass sich die in Abb. 79 dargestellte Situation ergibt.

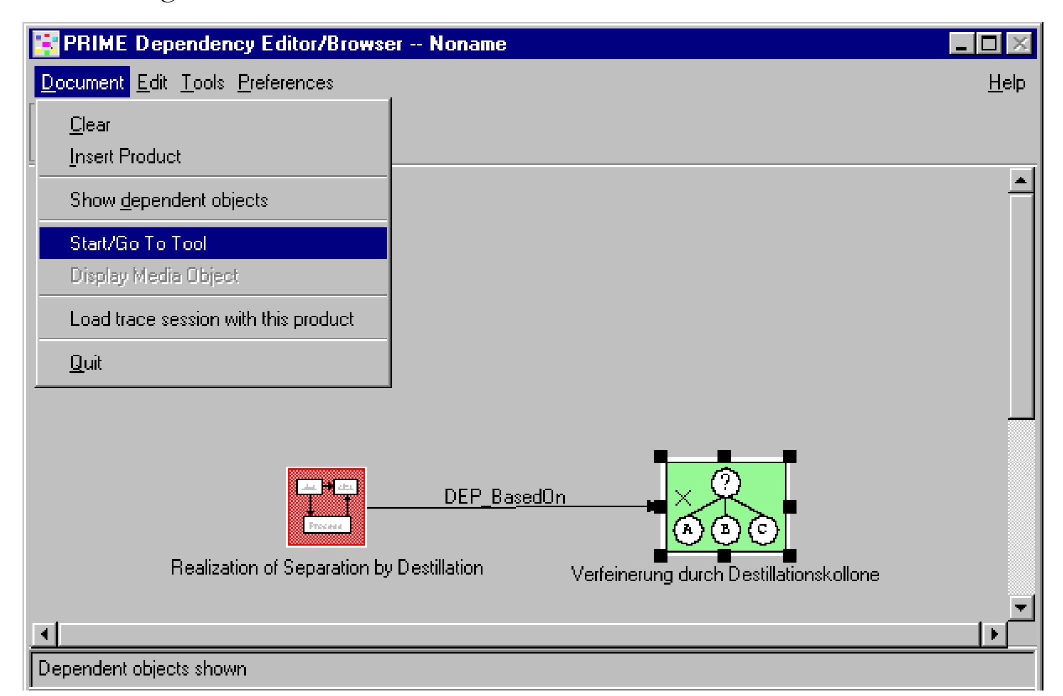


***Abb. 79**: Der Abhängigkeitseditor während der Ausführung des Entscheidungskontexts CC_SelectDepProduct*

Der bislang beschriebene Ablauf spielte sich automatisch im Hintergrund ab, ohne dass der Verfahrensingenieur die entsprechenden Aktionen selbstständig ausführen musste. Nun wird im Abhängigkeitseditor der Entscheidungskontext `CC_DEP_SelectDepObjects` aktiviert. In diesem Entscheidungskontext kann der Verfahrensingenieur entsprechend der Modellierung im M-Prozessmodell unter verschiedenen Optionen wählen. Beispielsweise kann der Verfahrenstechniker zu einem ausgewählten Objekt weiter entlang der Abhängigkeitsstruktur navigieren („Show Dependent Objects") oder sich die komplette Entstehungsgeschichte des Objekts als Prozessspur ansehen („Load Trace Session with this Product").

Der Verfahrensingenieur entscheidet sich dafür, das Objekt „Verfeinerung durch Destillationskolonne" genauer zu studieren und wählt dazu das Kommando „Start/Go to Tool". Da es sich bei dem ausgewählten Objekt um einen Entscheidungsobjekt handelt, aktiviert er mit diesen Interaktionen den Entscheidungskontext `CC_DEC_InspectDecision`. Abb. 80 zeigt den zu diesem Zeitpunkt erreichten Ausführungszustand im übergeordneten Plankontext `PC_InspectRelatedObjects`.

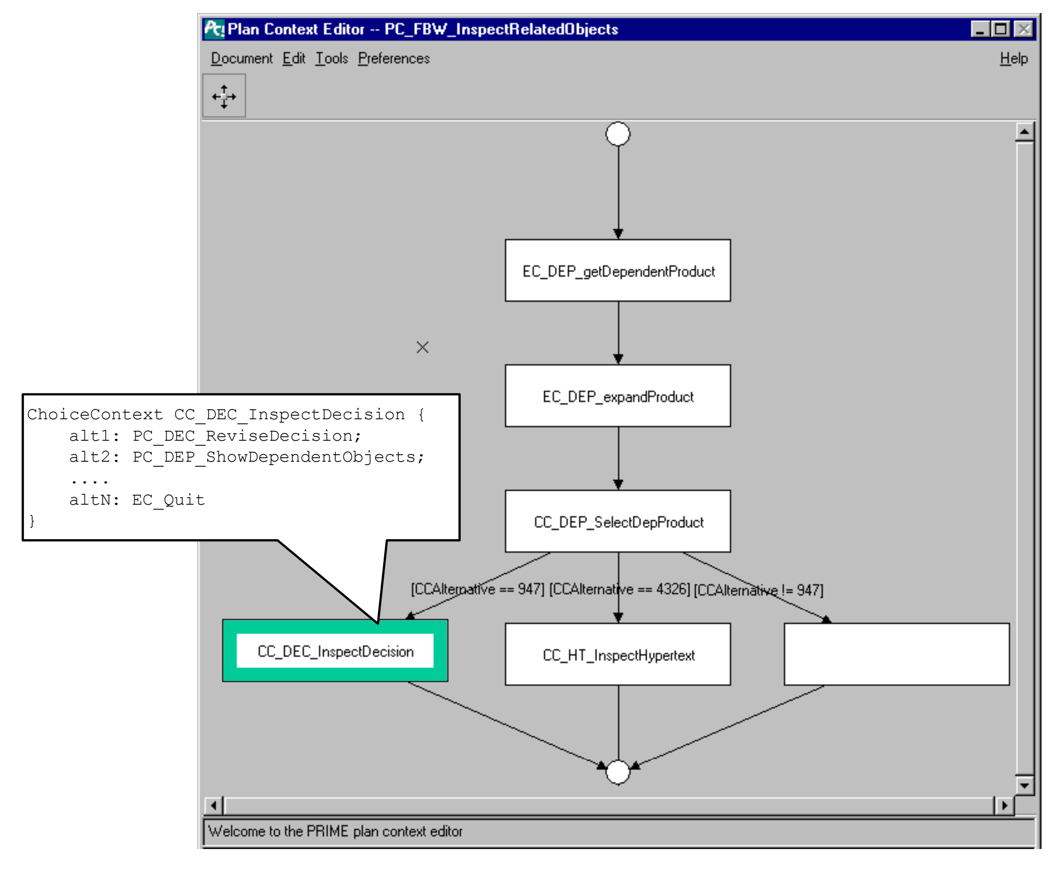


***Abb. 80**: Ausführung des eingebetteten Plankontexts PC_FBW_InspectRelatedObjects*

Die von der Prozessmaschine angeforderte Ausführung des Entscheidungskontexts CC_DEC_InspectDecision startet den Entscheidungseditor mit dem ausgewählten Entscheidungsobjekt. Dieser Entscheidungskontext konfiguriert den Entscheidungseditor so, dass der Benutzer das Entscheidungsdokument unverändert lassen oder die getroffene Entscheidung revidieren kann.

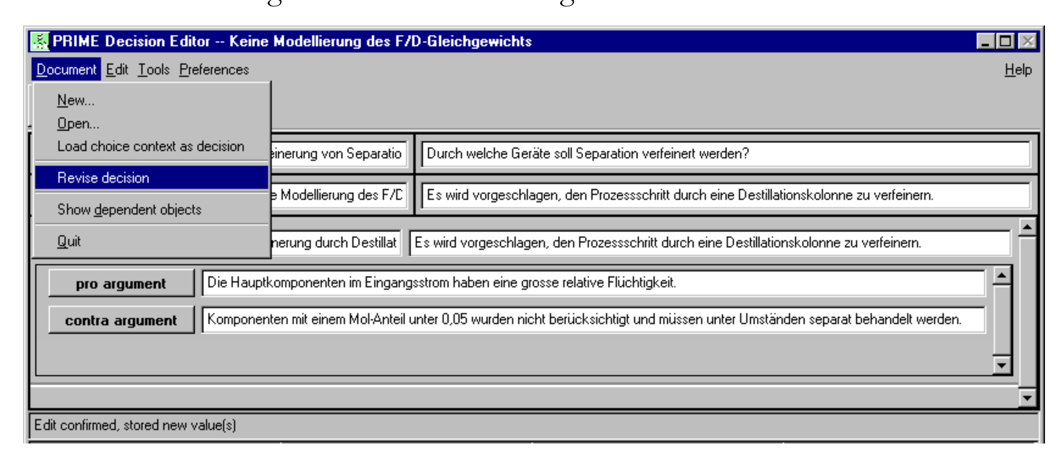


***Abb. 81**: Entscheidungsrevision im Entscheidungseditor*

Der Verfahrensingenieur erkennt, dass die bisherige Verfeinerung des VT-Prozessschritts Separation durch eine Destillationskolonne unter der Annahme geschah, dass sich in dem zu trennenden Stoffgemisch außer Wasser und CL keine weiteren nennenswerten Verunreinigungen befinden (siehe Abb. 81). Mittlerweile liegen dem Verfahrensingenieur jedoch die Ergebnisse von Laborexperimenten vor, die dieser Annahme widersprechen. Also beschließt er, die bisherige Entscheidung zu revidieren, und aktiviert den Plankontext PC_DEC_ReviseDecision. Dieser Plankontext stellt sicher, dass die Revision einer Entscheidung nachvollziehbar dokumentiert wird (hier nicht im Detail dargestellt).

Nach der Entscheidungsrevision geht die Kontrolle wieder auf den als Statechart modellierten Plankontext PC_FBW_InspectRelatedObjects und von dort auf den als SLANG-Netz modellierten Plankontext PC_RefineProcessDeviceWithDecision über.

Durch Aktivierung des Ausführungskontexts EC_FBW_RefineGroup wird im Fließbildeditor eine neue Verfeinerungsgruppe für den VT-Prozessschritt Separation angelegt, in die der Verfahrensingenieur nun eine Kaskade von Destillationskolonnen (statt einer einzelnen Kolonne) einträgt. Abschließend wird durch den Ausführungskontext EC_FBW_CreateDependencies die Abhängigkeitsstruktur so aktualisiert, dass sich die in Abb. 82 dargestellte Situation ergibt. Mit der Beendigung des Plankontexts geht die Werkzeugumgebung wieder in den unangeleiteten Unterstützungsmodus über.

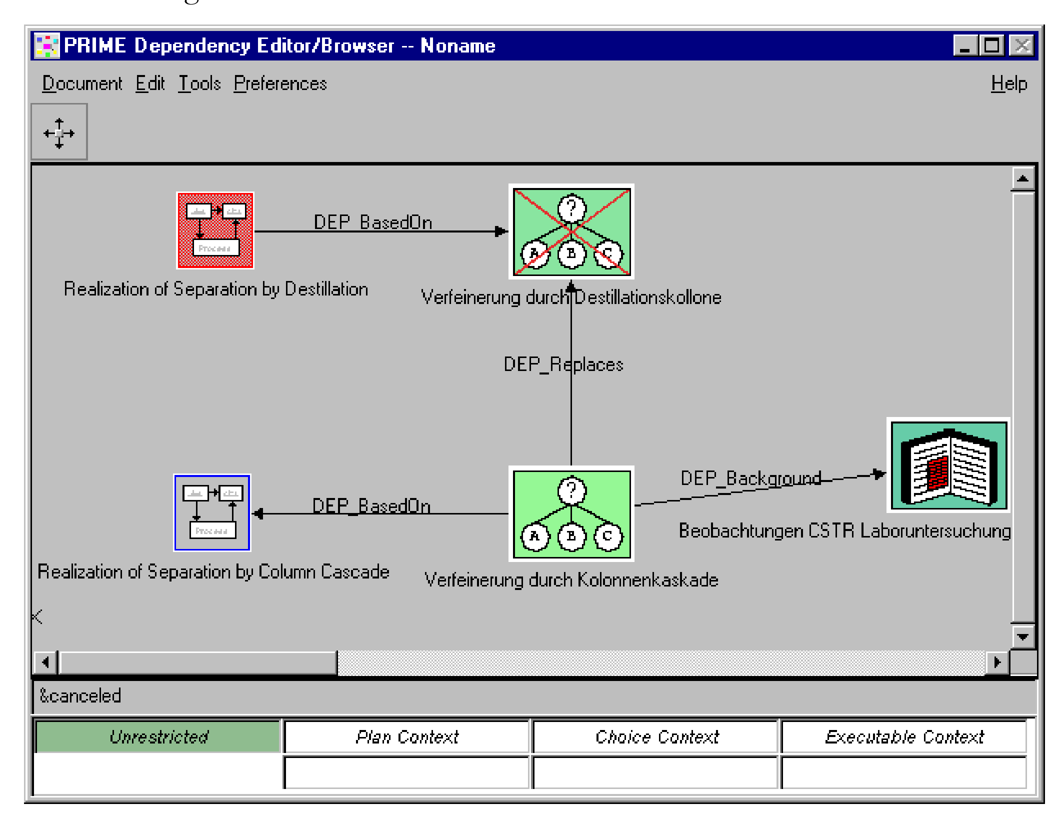


***Abb. 82**: Aktualisierte Abhängigkeitsstruktur nach der Entscheidungsrevision und Durchführung einer alternativen Verfeinerung*

8.3.3 Zusammenfassung

In diesem Abschnitt wurde anhand einer Beispielsitzung die Definition und Ausführung eines komplexen M-Prozessfragments illustriert. Gezeigt wurde ein M-Prozessfragment für die systematische Verfeinerung eines VT-Prozessschritts.

Verglichen mit einem unangeleiteten Verfeinerungsschritt hat der durch den Plankontext gesteuerte Verfeinerungsablauf den Vorteil, dass der Verfahrensingenieur (1) vorab auf bereits existierende Verfeinerungen hingewiesen wurde, (2) sich über die dafür vorliegenden Gründe informieren konnte und (3) bei der konsistenten Fortschreibung der Entscheidungs- und Abhängigkeitsdokumentation unterstützt wurde.

Für den Methodeningenieur wurde die M-Prozessfragmentdefinition dadurch erleichtert, dass er für Teilabläufe existierende M-Prozesskomponenten aus einer Komponentensammlung wiederverwenden konnte. Hierbei wurde die Einbettung

eines als UML-Statechart definierten Plankontexts in ein SLANG-Aktivitätsnetz demonstriert.

Über entsprechende Definitionen im Umgebungsmodell wurde der neue Plankontext dem Fließbildwerkzeug zugeordnet. Dies ermöglichte eine Aktivierung des M-Prozessfragmentes aus dem Fließbildwerkzeug heraus, ohne dass der Verfahrensingenieur seine gewohnte Arbeitsumgebung verlassen musste. Sichtbar wurde die M-Prozessunterstützung für den Verfahrensingenieur durch die Automatisierung von Teilabläufen und die Anpassung der auswählbaren Optionen in den Werkzeugen.

8.4 Fazit

Gegenstand dieses Kapitels waren Beispielanwendungen des PRIME-Rahmenwerks, mit denen wir seine Praktikabilität nachweisen konnten. Wir haben zunächst die Entwicklungshistorie des PRIME-Rahmenwerk ausgehend von der ursprünglichen PROART-Umgebung skizziert und sind dann näher auf die im Sonderforschungsbereich IMPROVE entstandene verfahrenstechnische Entwurfsumgebung PRIME-IMPROVE eingegangen. Von den bisherigen PRIME-Anwendungen war diese die bislang anspruchsvollste. Die besonderen Herausforderungen lagen hier in der Heterogenität der zu integrierenden Werkzeuge und in den Schnittstellen zu externen Unterstützungsfunktionalitäten. Anhand einer Beispielsitzung aus dem IMPROVE-Kontext haben wir den Umgang mit einer PRIME-basierten Umgebung aus Sicht des Methodeningenieurs und des Anwenders illustriert.

Kapitel 9

Schlussbetrachtungen

9.1 Beiträge der Arbeit

Gegenstand der vorliegenden Arbeit war ein Rahmenwerk für die Einbindung von Software-Werkzeugen in definierte Arbeitsprozesse. Ziel war die Unterstützung und flexible Anpassbarkeit der durch die Werkzeugumgebung unterstützten Arbeitsabläufe auf der Basis expliziter Prozess- und Werkzeugmodelle. Die dabei erzielten Ergebnisse lassen sich wie folgt zusammenfassen.

- *Vergleich von Prozessunterstützungsansätzen.* Wir haben zunächst ein Klassifikationsschema für die Bewertung existierender, prozessorientierter Unterstützungsfunktionen hinsichtlich der Merkmale *unterstützte Projektebene*, *Integrationstiefe*, *Kontextbezogenheit*, *Anpassbarkeit* und Abdeckung des Spektrums unterschiedlicher *Unterstützungsmodi* entwickelt. Anhand dieses Klassifikationsschemas haben wir die Unterstützungsleistung von Methodenhandbüchern, Hilfesystemen, Assistenten und Interface-Agenten sowie prozesszentrierten Umgebungen diskutiert sowie die Schwächen und Stärken der einzelnen Ansätze identifiziert. Aus dem Vergleich haben wir gefolgert, dass prozesszentrierte Umgebungen grundsätzlich die gerade in kreativen Entwurfsdomänen nötige Flexibilität hinsichtlich der Definition neuer Prozesse oder der Anpassung existierender Prozesse bieten, aber nicht die nötige Integrationstiefe mit der eigentlichen Werkzeugumgebung aufweisen.
- *Literaturüberblick zur Werkzeugintegration.* In einem umfassenden Literaturüberblick haben wir existierende Ansätze zur Lösung des Integrationsproblem einer kritischen Bewertung unterzogen. Zur Strukturierung der Literaturanalyse wurden zunächst insgesamt *sechs Kernanforderungen* herausgearbeitet, die den Übergang von einer prozess*zentrierten* Umgebungen hin zu prozess*integrierten* Umgebungen kennzeichnen: (1) Datenintegration zwischen den Prozessdomänen, (2) Prozessorientierte Mediation von Werkzeuginteraktionen, (3) Konzeptuelle Beschreibung von Werkzeugdiensten, (4) Synchronisation zwischen den Prozessdomänen, (5) prozesssensitive Benutzeroberflächen und (6) werkzeugunterstützter Aufruf von Prozessfragmenten. Die Betrachtung existierender prozesszentrierter Umgebungen sowie weiterer Ansätze aus den Bereichen Datenintegration, Kommunikationsinfrastrukturen, komponentenbasierte Softwareentwicklung, Werkzeugspezifikation, Prozessmodellierung, Benutzeroberflächen und Softwareergonomie ergab, dass zu Teilaspekten bereits mitunter ausgereifte Lösungsansätze vorliegen. Uns ist jedoch *keine* Lösung bekannt, die alle

genannten Anforderungen in einem ganzheitlichen Ansatz zusammenführt und umsetzt.

- *Integrierte Prozess- und Werkzeugmodellierung.* Der in dieser Arbeit vorgeschlagene Lösungsansatz zur umfassenden Umsetzung der genannten Anforderungen basiert auf der Grundidee, Wissen über Prozesse *und* Werkzeuge gleichberechtigt in konzeptuellen Modellen explizit zu erfassen und diese Modelle miteinander zu verzahnen. Für die fragmentweise Modellierung kreativer Arbeitsprozesse sind wir von einer existierenden Prozessmodellierungssprache, dem kontextbasierten NATURE-Prozessmodell, ausgegangen. Diese wurde ergänzt um eine Werkzeugmodellierungssprache, in der die Werkzeuge durch Definition ihrer Basisfunktionalitäten und ihres Interaktionsverhaltens spezifiziert werden können. Der modulare Aufbau des Gesamtmodells aus einem Prozess- und einem Werkzeugmodell ermöglicht eine saubere Trennung zwischen prozessrelevanten und werkzeuginhärenten Aspekten. Das heißt, dass Prozesswissen in Form von Kontextdefinitionen zunächst unabhängig von einer Beschreibung der Werkzeuge definiert werden kann und umgekehrt Werkzeuge prozessneutral beschrieben werden können. Erst durch die Zuordnung zwischen Prozess- und Werkzeugmodell innerhalb des so genannten *Umgebungsmodells* wird eine spezifische prozessintegrierte Umgebung konfiguriert.

- *Interoperabilität von Prozesssprachen.* Das NATURE-Prozessmodell kann nicht direkt als ausführbare Prozessmodellierungssprache verwendet werden kann, da es keine geeigneten Konzepte zur Formalisierung von Situationen und zur Festlegung eines Kontrollflusses in Plankontexten anbietet. Daher musste eine geeignete Einbettung eines Kontrollmodells in das NATURE-Prozessmodell gefunden werden. Um Flexibilität und Anpassbarkeit an domänenspezifische Bedürfnisse zu gewährleisten, haben wir das Ziel verfolgt, nicht nur einen Ablaufformalismus fest vorzugegeben oder einen neuen zu erfinden, sondern die Auswahl aus einer Palette unterschiedlicher Formalismen anzubieten, die für einen betrachteten Ablauf jeweils besonders geeignet sind. Hieraus ergab sich das Problem der *Interoperabilität* verschiedensprachlicher Prozessfragmente. Als Lösungsansatz haben wir uns für einen komponentenbasierten Modellierungsansatz entschieden, bei dem das NATURE-Prozessmodell so erweitert wurde, dass Kontexte als Komponenten mit definierten Schnittstellen modelliert und in einer Verwendungssprache unabhängig von ihrer Implementierung wiederverwendet werden können.

- *Implementierungs-Frameworks.* Das Konzept zur integrierten Werkzeug- und Prozessmodellierung wurde in Form zweier generischer, objektorientierter Frameworks umgesetzt. In der Durchführungsdomäne definiert das GARPIT-Framework die Grundstruktur eines prozessintegrierten Werkzeugs. Es stellt vorgefertigte Komponenten zur Synchronisation mit der Leitdomäne und zur Interpretation des Umgebungsmodells bereit und erlaubt die flexible Erweiterung um spezifische Werkzeugfunktionalitäten. Für die Integration existierender Werkzeuge wurde das GARPIT-Framework zu einem generischen Prozessintegrations-Wrapper modifiziert und anhand der Integration von insgesamt drei weit verbreiteten Fremdwerkzeugen validiert. In der Leitdomäne steht mit dem GARPEM-Framework ein generischer Rahmen für die Integration spezifischer Prozesssprachenintepreter

zur Verfügung. Das GARPEM-Framework wurde am Beispiel der Einbettung eines SLANG- und eines UML-Statechart-Interpreters validiert.

- *Validierung in mehreren Anwendungen.* Das PRIME-Rahmenwerk wurde zur Entwicklung mehrerer prozessintegrierter Umgebung verwendet und dabei erfolgreich validiert. Die gewonnenen Erfahrungen haben zur einer stetigen Weiterentwicklung des Rahmenwerks geführt. Von den bisherigen Anwendungen war die in dieser Arbeit detailliert beschriebene PRIME-IMPROVE-Umgebung die anspruchsvollste. Die besonderen Herausforderungen lagen hier in einer Einbeziehung existierender Werkzeuge, die einen unterschiedlichen Grad der Prozessintegrierbarkeit aufwiesen.

9.2 Erfahrungen und kritische Bewertung

Mit den entwickelten Werkzeug-Umgebungen wurden kleinere Nutzerstudien mit Studenten (PROART 2.0 und PRIME-CREWS) bzw. Verfahrenstechnik-Ingenieuren (PRIME-IMPROVE) durchgeführt. Die dabei gewonnenen Erfahrungen ermöglichen eine Bewertung des Ansatzes aus drei verschiedenen Blickwinkeln: denen des Anwenders, des Methodeningenieurs und des Werkzeugentwicklers.

9.2.1 Sicht des Anwenders

Aus Benutzersicht wurde die Anpassung des Werkzeugverhaltens an den aktuellen Prozesskontext und die dafür definierten Prozessmodelle als durchweg sehr hilfreich empfunden. Durch die automatische Anpassung der auswählbaren Kommandos und Produkte wurden die Anwender über die jeweils aktuell anwendbaren Prozessdefinitionen informiert und mussten diese nicht in extern vorliegenden Vorgehensbeschreibungen nachlesen. Auch das Ausblenden aktuell nicht relevanter Kommandos und Produkte wurde als positiv angesehen, da dadurch eine klare Fokussierung auf die zu bearbeitende Aufgabe erreicht wurde. Es wurde jedoch bemängelt, dass sich beim Wechsel eines Entscheidungskontexts in einem Werkzeug die Positionen der Menüeinträge laufend ändern und sich der Benutzer jeweils an ein neues Erscheinungsbild der Menüs gewöhnen muss.

Die nahtlose Integration der Prozessanleitung in die Werkzeug-Umgebung, d.h. die vereinheitlichte Aktivierung von werkzeuginternen Diensten, Diensten anderer Werkzeuge und Prozessfragmenten, wurde gegenüber der in prozesszentrierten Umgebungen und Workflow-Managementsystemen sonst üblichen Interaktion über einen separaten Agenda-Manager als großer Vorteil betrachtet. So erwähnten mehrere Benutzer, dass sie erst im nachhinein realisierten, dass sie gerade von einem Prozessfragment durch einen für sie unbekannten Ablauf geführt worden waren. In manchen Fällen fühlten sich die Benutzer nach der Aktivierung eines Plankontextes allerdings auch etwas orientierungslos. Hier erwies es sich als nachteilig, dass der Benutzer während der Ausführung eines Prozessfragments in seinen Werkzeugen jeweils nur den Überblick über die als nächstes möglichen Schritte erhält. Bei komplexeren Prozessfragmenten kann dies dazu führen, dass der Benutzer das Gesamtziel des Prozessfragments aus den Augen verliert und die Notwendigkeit einzelner Schritte bzw. deren Abfolge nicht mehr nachvollziehen kann. Die Möglichkeit, sich im Plankontexteditor und im Prozessspurenvisualisierer über den aktuellen Ausführungszustand zu informieren, wurde nur

von wenigen Benutzern genutzt, die sich vorher mit den entsprechenden Notationen vertraut gemacht hatten.

Natürlich wurde auch die Frage aufgeworfen, ob eine in die Werkzeuge integrierte Prozessunterstützung generell für eine bessere Qualität des Entwurfs sorgt oder umgekehrt die einem Entwurfsprozess inhärente Kreativität behindert. Hier wurde als positiv hervorgehoben, dass PRIME gar nicht erst den Versuch unternimmt, kreative Entwurfsprozesse durchgehend und präskriptiv anleiten zu wollen, sondern auf eine situative Unterstützung wohlverstandener Teilabläufe ausgelegt ist. Die Anleitung durch komplexe Prozessfragmente wurde insbesondere für die Präskription und Automatisierung der Aufzeichnung von Nachvollziehbarkeitsinformationen (z.B. Entscheidungsdokumentation) als sinnvoll angesehen.

9.2.2 Sicht des Methodeningenieurs

Die drei Kontexttypen unterstützten den Methodeningenieur bei der Strukturierung von Prozessmodellen, und zwar unabhängig von einer konkreten Prozessmodellierungssprache wie beispielsweise SLANG-Netze oder Statecharts. Im Unterschied zu den meisten anderen Modellierungsansätzen wurde der Methodeningenieur angehalten, Entscheidungen explizit zu definieren. Die Notwendigkeit, Werkzeugfunktionalitäten in Modellen zu spezifizieren, veranlasste die Modellierer, intensiver über die geeignete Granularität von Prozessschritten nachzudenken. Darüber hinaus halfen die Werkzeugmodelle dem Methodeningenieur, bei der Prozessmodelldefinition die vorhandene Werkzeugfunktionalität zu berücksichtigen. Die Definition von Werkzeug und Prozessmodellen sowie das integrierte Umgebungsmodell ermöglichten eine leichte Anpassung der von der Umgebung angebotenen Unterstützung. Besonders in experimentellen Anwendungen und für wenig verstandene Prozesse, in denen ständig neue Erfahrungen über verbesserte Vorgehensweisen gesammelt werden (wie in dem oben beschriebenen IMPROVE-Projekt), erwies sich die Aufwandsminimierung für die Anpassung der Unterstützung als essentiell.

9.2.3 Sicht des Umgebungsentwicklers

Die Realisierung von insgesamt 17 Werkzeugen mit einem Wiederverwendungsgrad von durchschnittlich über 85 % und die Einbettung zweier Prozesssprachen-interpreter belegen, dass sich die zusätzliche Investition in die Entwicklung der Implementierungs-Frameworks gelohnt hat. Die Werkzeugimplementierung wurde durch das GARPIT-Framework signifikant erleichtert, da wesentliche Teile der Architektur bereits als wiederverwendbare Komponenten vorlagen. Die klare Definition der Variationspunkte und die vollständige Kapselung des Kontrollflusses erlaubte ein einfaches Einklinken werkzeugspezifischer Funktionalität. Insbesondere brauchten sich die Entwickler nicht mehr um die schwierige architekturelle und technische Integration zwischen Modellierungs-, Leit- und Durchführungsdomäne zu kümmern, da diese bereits vollständig auf Framework-Ebene vorweggenommen ist.

Es zeigte sich, dass die Entwickler die Funktionalität eines Werkzeuges problemlos erweitern konnten, ohne sich in die Struktur dieses Werkzeuges intensiv einarbeiten zu müssen bzw. es selbst implementiert zu haben. Natürlich erfordert ein Framework einen zusätzlichen Lernaufwand, bevor es sinnvoll für die Ent-

wicklung eines konkreten Werkzeugs eingesetzt werden kann. Es stellte sich jedoch heraus, dass dieser Zusatzaufwand relativ moderat ausfällt. Studentische Hilfskräfte und Diplomanden, die zum ersten Mal mit dem PRIME-Framework konfrontiert wurden, waren je nach Programmiererfahrung nach ein bis zwei Wochen Einarbeitungszeit in der Lage, eigenständig ein Werkzeug zu implementieren oder ein bestehendes zu erweitern.

Weiterhin zeigte sich, dass die Werkzeugmodellierungskonzepte und deren Verknüpfung zu Ausführungs- und Entscheidungskontexten zur Definition prozessintegrierter Werkzeuge grundsätzlich ausreichen. Die Definition komplexer Abläufe innerhalb des Prozessmodells zwang die Werkzeugentwickler, Prozesswissen explizit zu machen und so das „Prozess-im-Werkzeug"-Syndrom [Mont94] zu vermeiden. Die Kontextgebundenheit von Benutzereingaben bei Entscheidungskontexten reduziert mögliche Fehleingaben des Benutzers auf ein Minimum, so dass nur sehr wenige Konsistenzchecks explizit ausprogrammiert werden mussten.

Als nicht ganz unproblematisch erwies sich jedoch die Festlegung der „richtigen" Granularität der elementaren Werkzeugaktionen, die als atomare Bausteine auf der Prozessmodellierungsebene miteinander verschaltet werden können. Hier war bisweilen die Tendenz zu beobachten, den kompletten Methodenvorrat des internen, werkzeugspezifischen Produktmodell in Form von Aktionen der Prozessmodellierung zugänglich zu machen, um höchstmögliche Adaptabilität auf Prozessmodellierungsebene zu garantieren. Bei einer derart feinen Granularität besteht jedoch die Gefahr, dass ein großer Teil der Werkzeugprogrammierung auf die Ebene der Prozessmodellierung verlagert wird. Prozessmodellierung würde dadurch zum Werkzeug-„Skripting" degenerieren und in die Nähe der Makroprogrammierung geraten, wie sie etwa innerhalb der Microsoft-Office-Familie mittels VBA (Visual Basic for Applications) möglich ist. In diesem Fall ließe sich ein Werkzeug zwar nahezu beliebig manipulieren, jedoch würden hier Aspekte der Organisation von Arbeitsabläufen zunehmend durch eher programmiertechnische Aspekte des Verschaltens von Werkzeugfunktionalitäten überlagert. Als Schlussfolgerung sollten also die für die Prozessmodellierung relevanten Aktionen eines Werkzeugs mit den in der Modellierungsdomäne betrachteten kleinsten Arbeitseinheiten auf Prozessebene korrespondieren. Die Kunst bestand dann darin, die Granularität der Basisdienste auf möglichst hohem semantischem Niveau gerade so fein zu wählen, dass deren interne Ablauflogik nicht mehr prozessrelevant ist, d.h. nicht auf Prozessmodellierungsebene adaptabel sein muss. Dass es hierfür kein Patentrezept geben kann, ist einsichtig. Vielmehr mussten hier Kompromisse eingegangen werden, die den Trade-Off zwischen maximaler Flexibilität bei der Ausgestaltung von Werkzeugfunktionalitäten einerseits und Klarheit und Einfachheit der Prozessmodelle andererseits berücksichtigten.

Trotz der genannten Einschränkungen konnte mit der Implementierung des PRIME-Rahmenwerkes und seiner Beispielanwendungen die Praktikabilität des in dieser Arbeit entwickelten Adaptabilitäts- und Prozessunterstützungsansatzes nachgewiesen werden.

9.3 Ausblick

Zum Abschluss wollen wir noch einige sinnvolle Ergänzungen und Erweiterungsmöglichkeiten des vorgestellten Ansatzes aufzeigen:

- *Werkzeugübergreifender Situationsbegriff.* Die Ausgestaltung des Situationsbegriffs ist in der aktuellen Implementierung nicht sehr ausdrucksstark. Neben der geringen Mächtigkeit der Situationssprache liegt dies vor allem daran, dass die Situationsauswertung jeweils werkzeuglokal erfolgt. Durch eine erweiterte, werkzeugübergreifende Situationsanalyse ließen sich auch komplexe Produktkonstellationen charakterisieren, die über mehrere Werkzeuge verteilt vorliegen. Mit der erweiterten, zweistufigen Situationsanalyse unter Rückgriff auf eine Prozessdatenbank, wie sie im Fließbildwerkzeug realisiert wurde, konnte bereits ein erster viel versprechender Schritt in diese Richtung aufgezeigt werden, der jedoch noch systematischer untersucht werden muss.
- *Generierung von Werkzeugdiensten.* Die Implementierungs-Frameworks entlasten den Umgebungsentwickler von der architekturellen und technischen Integration der Prozessdomänen, während die werkzeugspezifische Funktionalität jedoch noch weitgehend von Hand ausprogrammiert werden muss. Eine weitere Aufwandsminimierung könnte die (semi-)automatische Generierung von Werkzeugdiensten auf Basis eines verfeinerten Werkzeugsmodells bringen. Erste konzeptuelle Grundlagen wurden bereits in einem Kooperationsprojekt mit einer Forschergruppe an der Universität Jyväskylä erarbeitet. Hier wurden Möglichkeiten und Grenzen einer Integration zwischen dem PRIME-Rahmenwerk und der CASE-Shell MetaEdit+ diskutiert [Lyy*98].
- *Verzahnung zwischen Prozessdefinition und -ausführung.* In den bisherigen PRIME-Anwendungen wurde die verschränkte Modellierung und Ausführung von Prozessfragmenten noch nicht systematisch untersucht. Hierfür liegen jedoch bereits alle technischen Voraussetzungen vor, da die Administrations- und Metamodellierungswerkzeuge ebenso wie alle anderen PRIME-Werkzeuge prozessintegriert sind. Es wäre also denkbar, den Anwender bei der ad-hoc-Definition neuer oder der Änderung existierender Prozessfragmente durch geeignete Meta-Prozessfragmente zu unterstützen.
- *Abgeschwächte Formen der Prozessintegration.* Die Prozessintegration im a posteriori-Kontext ist erst für den Fall voll prozessintegrierbarer Werkzeuge vollständig verstanden. Bei nur eingeschränkt integrierbaren Werkzeugen sind insbesondere Aspekte der Synchronisation zwischen Prozessanleitung und –ausführung und der Sicherung der Nachvollziehbarkeit noch weitgehend ungeklärt. Diesen Fragestellungen wird zur Zeit in der zweiten Förderperiode des SFB IMPROVE nachgegangen.

In den in dieser Arbeit betrachteten Anwendungsfällen liegen hinsichtlich der Akzeptanz und Qualität der Benutzerführung durch prozessintegrierte Werkzeuge bislang nur qualitative Aussagen aus ersten Nutzungsexperimenten vor. Eine wissenschaftlich fundierte Evaluierung kann nur im Rahmen einer empirischen Untersuchung erfolgen, in der erfahrene Entwickler mit einer PRIME-Umgebung ein anwendungsnahes Problem bearbeiten. Mittlerweile haben die Werkzeuge der PRIME-Umgebung einen Grad der Robustheit erreicht, die größer angelegte Nutzerexperimente sinnvoll erscheinen lassen. Hierzu werden (ebenfalls im Rahmen des SFB IMPROVE) zur Zeit in Zusammenarbeit mit dem Aachener Lehrstuhl und Institut für Arbeitswissenschaften entsprechende Evaluationsstudien vorbereitet.

Insgesamt haben uns jedoch auch die bereits jetzt vorliegenden Erfahrungen darin bestärkt, dass eine an Prozessen ausgerichtete Adaptabilität von Werkzeugumgebungen dem Anwender viele Vorteile bringt und in Zukunft noch an Bedeutung gewinnen wird. Dieser Trend wird zum Beispiel auch aktuell durch die Tatsache belegt, dass Microsoft in seiner kommenden Betriebssystem-Version Windows XP verstärkt auf eine aufgabenorientierte Benutzerführung setzt.

Literaturverzeichnis

[AbAG95] Abowd, G., Allan, R. und Garlan, D.: *Formalizing Style to Understand Descriptions of Software Architecture*. ACM Transactions on Software Engineering and Methodology, 4(4): S. 319-364, 1995.

[AbCa96] Abadi, M. und Cardelli, L.: *A Theory of Objects*. Springer Verlag, New York, 1996.

[Abel95] Abeln, O.: *CAD-Referenzmodell*. B.G. Teubner Verlagsgesellschaft, Stuttgart, 1995.

[ABGM93] Armenise, P., Bandinelli, S., Ghezzi, C. und Morzenti, A.: *A Survey and Assessment of Software Representation Formalisms*. International Journal of Software Engineering and Knowledge Engineering, 3(3), S. 410-426, 1993.

[Alde91] Alderson, A.: *Meta-CASE Technology*. In: Proceedings of the European Symposion on Software Development Environments and CASE Technology, Springer-Verlag, LNCS 509, 1991, S. 81-91.

[Alla97] Allan, R.: *A Formal Approach to Software Architecture*. Tech. Report, Carnegie Mellon University, CMU/CS-97-144, 1997.

[Alo*96] Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Guenthoer, R. und Mohan, C.: *Advanced Transaction Models in Workflow Contexts*. In: Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, USA, IEEE Computer Society Press, 1996, S. 574-583.

[AmCF97] Ambriola, V., Conradi, R. und Fuggetta, A.: *Assessing Process-Centered Software Engineering Environments*. ACM Transactions of Software Engineering and Methodology, 6(3): S. 283-328, July 1997.

[And*99] Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H.-J., Kuske, S., Plump, D., Schürr, A. und Taenzer, G.: *Graph Transformation for Specification and Programming*. Science of Computer Programming, 34(1): S. 1-54, 1999.

[Ande90] Andersen, O.: *The Use of Software Engineering Data in Support of Project Management*. Software Engineering Journal, 5(6): S. 250-256, 1990.

[AnGr94] Anderson, M. und Griffiths, P.: *The Nature of the Software Process Modelling Problem is Evolving*. In: Proc. 2nd European Workshop on Software Process Technology (EWSPT '94), Villard de Lans, France, 1994, S. 31-34.

[ApRi99] Appelrath, H.-J. und Ritter, N.: *R/3-Einführung - Methoden und Werkzeuge*. Springer-Verlag Berlin Heidelberg, 1999.

[ArOq94] Arbaoui, S. und Oquendo, F.: *PEACE: Goal-Oriented Logic-Based Formalism for Process Modelling*. In: Finkelstein, A., Kramer, J. und Nuseibeh, B. (Hrsg.): Software Process Modelling and Technology. Research Studies Press, 1994, S. 249-278.

[AT&T84] AT&T: *UNIX System V Documenter's Workbench - Introduction and Reference Manual*. 1984.

[AT&T96] AT&T, DSTC, DEC, HP, ICL, Nortel und Novell: *Trading Object Service*. OMG RFP5 Submission orbos/96-05-06, Version 1.0, 1996.

[Atk*89] Atkinson, M.P., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D. und Zdonik, St.: *The Object-Oriented Database System Manifesto*. In: Kim, W., Nicolas, J.-M. und Nishio, Sh. (Hrsg.), Deductive and Object-Oriented Databases, Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto Research Park, Kyoto, Japan, North-Holland/Elsevier Science Publishers, 1989, S. 223-240.

[AvBC96] Avrilionis, D., Belkhatir, N. und Cunin, P.: *A Unified Framework for Software Process Enactment and Improvement*. In: Proceedings of the 4th International Conference on the Software Process (ICSP4), 1996.

[AvBC96a] Avrilionis, D., Belkhatir, N. und Cunin, P.: *Improving Software Process Modelling and Enactment Techniques*. In: Montangero, C. (Hrsg.), Proceedings of the 5th European Workshop on Software Process Technology (EWSPT '96), Springer Verlag, LNCS 1149, 1996, S. 65-74.

[AvFi88] Avison, D.E. und Fitzgerald, G.: *Information Systems Development: Methodologies, Techniques, and Tools*. Blackwell, Oxford, UK, 1988.

[BaCK98] Bass, L., Clements, P. und Katzman, R.: *Software Architecture in Practice*. Addison Wesley, Reading, 1998.

[BaCR94] Basili, V.R., Caldiera, G. und Rombach, D.: *Goal Question Metric Paradigm*. In: Marciniak, J. (Hrsg.): Encyclopedia of Software Engineering, Vol. 1. John Wiley & Sons, 1994, S. 528-532.

[BaDF96] Bandinelli, S., Di Nitto, E. und Fuggetta, A.: *Supporting Cooperation in the SPADE-1 Environment*. IEEE Transactions on Software Engineering, 22(12): S. 841-865, Dec. 1996.

[BaFG93] Bandinelli, S., Fuggetta, A. und Ghezzi, C.: *Software Process Model Evolution in the SPADE Environment*. IEEE Transactions on Software Engineering, 19(12): S. 1128-1144, Dec. 1993.

[BaGl98] Bauer, Ch. und Glasson, B.: *Extending the Concept of a Reference Model across Industries*. In: Proceedings IFIP WG 8.2 & WG 8.6 Joint Working Conference on Information Systems: Current Issues and Future Changes, Helsinki, Finnland, 1998, S. 471-488.

[BaKa91] Barghouti, N. und Kaiser, G.E.: *Scaling Up Rule-Based Software Development Environments*. In: Proc. 3rd European Software Engineering Conference, ESEC '91, Milan, Italy, Springer, LNCS 550, 1991, S. 380-395.

[BaKr93] Barghouti, N. und Krishnamurthy, B.: *An Open Environment for Process Modeling and Enactment*. In: Proc. 8th Intl. Software Process Workshop: State of the Practice in Process Technology, Wadern, Germany, March 1993, S. 33-36.

[BaKr95] Barghouti, N. und Krishnamurthy, B.: *Using Event Contexts and Matching Constraints to Monitor Software Processes*. In: Proc. 17th Intl. Conf. on Software Engineering, Seattle, Washington, USA, May 1995, S. 83-92.

[BAKR95a] Bellissard, L., Atallah, S.B., Kerbrat, A. und Riveill, M.: *Component-based Programming and Application Management with Olan*. In: Briot, J. und Geib, J. (Hrsg.), Proceedings of Workshop on Object-Based Parallel and Distributed Computation, Springer-Verlag, LNCS 1107, 1995, S. 290-309.

[BaLi96] Banavar, G. und Lindstrom, G.: *An Application Framework for Module Composition Tools*. In: Proc. of the European Conference on Object-Oriented Programming, ECOOP '96, Springer-Verlag, LNCS 1098, 1996.

[Balz96] Balzert, H.: *Lehrbuch der Software-Technik - Software-Entwicklung*. Spektrum Akademischer Verlag, Heidelberg Berlin Oxford, 1996.

[BaMa98] Baumeister, M. und Marquardt, W.: *The Chemical Engineering Data Model VeDa. Part 1: The Data Definition Language VDDL*. Tech. Report, Lehrstuhl für Prozesstechnik, RWTH Aachen, TR 1998-01, 1998.

[Barg92] Barghouti, N.: *Supporting Cooperation in the MARVEL Process-Centered SDE*. In: Proc. 5th ACM SIGSOFT Symposium on Software Development Environments, SIGSOFT Notes, 1992, S. 21-31.

[BaRo88] Basili, V.R. und Rombach, H.D.: *The TAME Project: Towards Improvement-Oriented Software Environments*. IEEE Transactions on Software Engineering, 14(6): S. 758-773, 1988.

[BaSc88] Bauer, J. und Schwab, T.: *Anforderungen an Hilfesysteme*. In: Einführung in die Software-Ergonomie. de Gruyter-Verlag, Berlin, 1988, S. 197-214.

[BBFL94] Bandinelli, S., Braga, M., Fuggetta, A. und Lavazza, L.: *The Architecture of the SPADE-1 Process-Centered SEE*. In: Proc. 2nd European Workshop on Software Process Technology (EWSPT '94), Villard de Lans, France, 1994, S. 15-30.

[BCTW96] Barrett, D., Clarke, L., Tarr, P.L. und Wise, A.: *A Framework for Event-Based Software Integration*. ACM Transactions of Software Engineering and Methodology, 5(4): S. 378-421, Oct. 1996.

[Bec*99] Becker-Kornstaedt, U., Hamann, D., Kempkens, R., Rösch, P., Verlage, M., Webby, R. und Zettel, J.: *Support for the Process Engineer - The Spearmint Approach to Software Process Definition and Process Guidance*. In: Proc. CAiSE 99, 1999, S. 119-133.

[BeDa94] Bernstein, Ph. und Dayal, U.: *An Overview of Repository Technology.* In: Proc. 20th Intl. Conf. on Very Large Data Bases (VLDB '94), Santiago de Chile, Chile, Sept. 12-15 1994, S. 705-713.

[Beer90] Beeri, C.: *A Formal Approach to Object Oriented Databases.* Data Knowledge Engineering, 4(5): S. 353-382, 1990.

[BeKa98] Ben-Shaul, I.Z. und Kaiser, G.E.: *Federating Process-Centered Environments: The Oz Experience.* Automated Software Engineering Journal, 5(1): S. 97-132, 1998.

[BeKl96] Bergstra, J. und Klint, P.: *The ToolBus Coordination Architecture.* In: Ciancarini, P. und Hankin, C. (Hrsg.), Coordination Languages and Models, Springer-Verlag, LNCS 1061, 1996, S. 75-88.

[BeMS99] Berthold, A., Mende, U. und Schuster, H.: *SAP Business Workflow - Konzept, Anwendung, Entwicklung.* Addison-Wesley, 1999.

[BeMü99] Becker, J. und zur Mühlen, M.: *Towards a Classification Framework for Application Granularity in Workflow Management Systems.* In: Jarke, M. und Oberweis, A. (Hrsg.), Proc.of the 11th International Conference on Advanced Information Systems Engineering (CAiSE '99), Heidelberg, Springer Verlag, LNCS 1626, 1999, S. 411-416.

[Ber*99] Bernstein, Ph., Bergstraesser, Th., Carlson, J., Pal, S., Sanders, P. und Shutt, D.: *Microsoft Repository Version 2 and the Open Information Model.* Information Systems, 24(2): S. 71-98, 1999.

[BeRS99] Becker, J., Rosemann, M. und Schütte, R. (Hrsg.): *Referenzmodellierung - State-of-the-Art und Entwicklungsperspektiven.* Physika-Verlag Heidelberg, 1999.

[Bey*00] Beydeda, S., Gruhn, V., Schneider, C., Alloui, I., Oquendo, F. und Cimpan, S.: *Advanced Services for Process Evolution: Monitoring and Decision Support.* In: Proceedings of the 7th European Workshop on Software Process Technology, Kaprun, Österreich, Springer-Verlag, LNCS 1780, 2000.

[BFGL94] Bandinelli, S., Fuggetta, A., Ghezzi, C. und Lavazza, L.: *SPADE: An Environment for Software Process Analysis, Design, and Enactment.* Tech. Report, GOODSTEP Project, TR No. 020, 1994.

[Bias91] Bias, R.: *Walkthroughs: Efficient Collaborative Testing.* IEEE Software, 8(5): S. 94-95, 1991.

[BiJo87] Birman, K.P. und Joseph, T.A.: *Reliable Communication in Presence of Failure.* ACM Transactions on Computer Systems, 5(1): S. 47-76, 1987.

[Blas97] Blass, E.: *Entwicklung verfahrenstechnischer Prozesse - Methoden, Zielsuche, Lösungssuche, Lösungsauswahl.* Springer Verlag Berlin Heidelberg, 1997.

[Ble*99] Bleek, W.-G., Gryczan, G., Lilienthal, C., Lippert, M., Roock, S., Wolf, H. und Züllighoven, H.: *Frameworkbasierte Anwendungsentwicklung (Teil 2): Die Konstruktion interaktiver Anwendungen.* OBJEKTspektrum, 2/99: S. 78-83, 1999.

[Ble*99a] Bleek, W.-G., Lippert, M., Roock, S., Strunk, W. und Züllighoven, H.: *Frameworkbasierte Anwendungsentwicklung (Teil 3): Die Anbindung von Benutzeroberflächen und Entwicklungsumgebungen an Frameworks.* OBJEKTspektrum, 3/99: S. 90-95, 1999.

[BMCJ98] Böhm, M., Meyer-Wegener, K., Cap, C. und Jablonski, St.: *Ein konstruktiver Ansatz zur systematischen Entwicklung von Ausführungsanweisungen von Workflows.* Tech. Report, Technische Universität Dresden, TUD/FI/98-05-April, 1998.

[Boeh84] Boehm, B.: *Software Engineering Economics.* IEEE Transactions on Software Engineering, 10(1): S. 4-21, 1984.

[Boeh88] Boehm, B.: *A Spiral Model of Software Development and Enhancement.* IEEE Computer, 21(5): S. 61-72, 1988.

[Böhm98] Böhm, M.: *Integration externer Applikationen im Workflow-Management.* Informatik/Informatique, April 98: S. 23-27, 1998.

[BoJR99] Booch, G., Jacobson, I. und Rumbaugh, J.: *The Unified Modeling Language User Guide.* Adidson Wesley, 1999.

[Booc94] Booch, G.: *Object Oriented Design with Applications.* Benjamin/Cummings Publishing Company Inc., 1994.

[BoTa96] Bolcer, G.A. und Taylor, R.: Endeavors: *A Process System Integration Infrastructure*. In: Proceedings of the 4th International Conference on the Software Process, Brighton, Großbritannien, 1996, S. 76-89.

[Bran01] Brandt, S.: *Prozessintegration von Fremdwerkzeugen am Beispiel der verfahrenstechnischen Entwufsumgebung PRIME-IMPROVE*. Diplomarbeit, RWTH Aachen, 2001 (in Vorbereitung).

[BrDr95] Bröhl, A.-P. und Dröschel, W. (Hrsg.): *Das V-Modell*. Oldenbourg Verlag München Wien, 1995.

[Bre*93] Breitbart, Y., Deacon, A., Schek, H.-J., Sheth, A. und Weikum, G.: *Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows*. ACM SIGMOD Record, 22(3): S. 23-30, 1993.

[BrEM92] Brown, A., Earl, A. und McDermid, J.: *Software Engineering Environments - Automated Support for Software Engineering*. McGraw-Hill, 1992.

[BrEW92] Brown, A., Earl, A. und Wallnau, K.: *Past and Future Models of CASE Integration*. In: Proc. 5th Intl. Workshop on CASE (CASE '92), Montreal, Canada, July 1992, S. 36-45.

[BrFe92] Brown, A. und Feiler, P.: *An Analysis Technique for Examining Integration in a Project Support Environment*. In: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, Washington DC, USA, 1992, S. 139-148.

[Brin96] Brinkkemper, S.: *Method engineering: engineering of information systems development methods and tools*. Information and Software Technology, 38(4): S. 275-280, 1996.

[BrLW96] Brinkkemper, S., Lyytinen, K. und Welke, R. (Hrsg.): *Method Engineering - Principles of Method Construction and Tool Support*. Chapman & Hall , 1996.

[BrMc91] Brown, A. und McDermid, J.: *On integration and reuse in a software development environment*. In: Long, F. (Hrsg.): Software Engineering Environments. Ellis Horwood, 1991, S. 171-194.

[Bro*94] Brown, A., Carney, D.J., Morris, E., Smith, D. und Zarrella, P.: *Principles of CASE Tool Integration*. Oxford University Press, 1994.

[Broc95] Brockschmidt, K.: *Inside OLE*. Microsoft Press, 1995.

[Brow93] Brown, A.: *Control Integration through Message-passing in a Software Development Environment*. Software Engineering Journal, S. 121-131, May 1993.

[BrPS98] Bray, T., Paoli, J. und Sperberg-McQueen, C.M. (Hrsg.): *Extensible Markup Language (XML) 1.0*. Tech. Report, W3C, REC-xml-19980210, Feb. 1998. (http://www.w3.org/TR/REC-xml)

[Bus*96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. und Stal, M.: *Pattern Oriented Software Architecture - A System of Patterns*. Wiley & Sons, 1996.

[Byrn96] Byrne, B.: *IRDS Systems and Support for Present and Future CASE Technology*. In: Proceedings of CAiSE '96 DC, W4, Heraklion, Greece, 1996

[CaFM99] Casati, F., Fugini, M. und Mirbel, I.: *An Environment for Designing Exceptions in Workflows*. Information Systems, 24(3): S. 255-273, 1999.

[Caga90] Cagan, M.R.: *The HP SoftBench Environment: An Architecture for a New Generation of Software Tools*. Hewlett-Packard Journal, 42(3): S. 36-47, 1990.

[Car*93] Carr, M.J., Konda, S.L., Monarch, I., Ulrich, F.C. und Walker, C.F.: *Taxonomy-based Risk Identification*. Tech. Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania, USA, 1993.

[Cat*00] Catell, R., Barry, D., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T. und Velez, F (Hrsg.).: *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, 2000.

[CaWe85] Cardelli, L. und Wegner, P.: *On Understanding Types, Data Abstraction, and Polymorphism*. Computing Surveys, 17(4): S. 471-522, 1985.

[CDFG96] Cugola, G., Di Nitto, E., Fuggetta, A. und Ghezzi, C.: *A Framework for Formalizing Inconsistencies and Deviations in Human-Centered Systems*. ACM Transactions of Software Engineering and Methodology, 5(3): S. 191-230, 1996.

[CDGM95] Cugola, G., Di Nitto, E., Ghezzi, C. und Mantione, M.: *How To Deal With Deviations During Process Model Enactment*. In: Proc. 17th Intl. Conf. on Software Engineering, Seattle, Washington, USA, May 1995, S. 265-273.

[Chen76] Chen, P.P.S.: *The Entity-Relationship Approach: Towards a Unified View of Data*. ACM Transactions on Database Systems 1(1), 1976.

[ChHJ93] Chen, P.S., Hennicker, R., Jarke, M.: *On The Retrieval of Reusable Components*. In: Advances in Software Reuse, Selected Papers from the Second International Workshop on Software Reusability, Lucca, Italy, March 24-26, 1993, S. 99-108.

[ChNo92] Chen, M. und Norman, R.: *A Framework for Integrated CASE*. IEEE Computer, S. 18-22, March 1992.

[Chr*97] Christie, A., Levine, L., Morris, E., Riddle, W., Zubrow, D., Belton, T., Proctor, L., Cordelle, D., Ferotin, J.-E. und Solvay, J.-P.: *Software Process Automation: Interviews, Survey, and Workshop Results*. Tech. Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania, USA, CMU/SEI-97-TR-008, 1997.

[Chu*98] Chung, P.E., Huang, Y., Yajnik, S., Liang, D., Shih, J., Wang, C.-Y. und Wang, Y.-M.: *DCOM and CORBA: Side by Side, Step by Step, and Layer by Layer*. C++ Report, 10(1): S. 18-28, 1998.

[Clem96] Clements, P.: *A Survey of Architecture Description Languages*. In: Proceedings of the 8th International Workshop on Software Specification and Design, Paderborn, Germany, IEEE Computer Society Press, Los Alamitos, CA, 1996, S. 16-25.

[ClOs90] Clemm, G. und Osterweil, L.: *A Mechanism for Environment Integration*. ACM Transactions on Programming Languages and Systems, 12(1): S. 1-25, 1990.

[CNWL00] Conradi, R., Nguyen, M.N., Wang, A.I. und Liu, C.: *Planning Support to Software Process Evolution*. International Journal of Software Engineering and Knowledge Engineering, 10(1): S. 31-47, 2000.

[CoBe88] Conklin, J. und Begeman, M.: *gIBIS: A Hypertext Tool for Exploratoy Policy Discussion*. ACM Transactions on Office Information Systems, 6(4): S. 303-331, Oct. 1988.

[CoJa99] Conradi, R. und Jaccheri, M.L.: *Process Modelling Languages*. In: Derniame, J.-C., Kaba, B.A. und Wastell, D. (Hrsg.): Software Process: Principles, Methodology, and Technology. Springer Verlag, Berlin-Heidelberg, LNCS 1500, 1999, S. 27-52.

[CoSc95] Coplien, J. und Schmidt, D.C. (Hrsg.): *Pattern Languages of Program Design*. Addison-Wesley, Reading, 1995.

[CoWe98] Conradi, R. und Westfechtel, B.: *Version Models for Software Configuration Management*. ACM Computing Surveys, 30(2): S. 232-282, 1998.

[CoYo91a] Coad, P. und Yourdon, E.: *Object-Oriented Analysis*. Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[CoYo91b] Coad, P. und Yourdon, E.: *Object-Oriented Design*. Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[CuDF98] Cugola, G., Di Nitto, E. und Fuggetta, A.: *Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems*. In: Proc. 20th Intl. Conf. on Software Engineering, Kyoto, Japan, IEEE Computer Society Press, Aug. 1998, S. 261-270.

[Cugo98] Cugola, G.: *Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models*. IEEE Transactions on Software Engineering, 24(11): S. 982-1001, Nov. 1998.

[CuKO88] Curtis, B., Kellner, M. und Over, J.: *A Field Study of the Software Design Process for Large Systems*. Communications of the ACM, 33(11): S. 1268-1287, 1988.

[CuKO92] Curtis, B., Kellner, M. und Over, J.: *Process Modeling*. Communications of the ACM, 35(9): S. 75-90, Sept. 1992.

[DaEA98] Dami, L., Estublier, J. und Amiour, M.: *APEL: A Graphical yet Executable Formalism for Process Modelling*. Automated Software Engineering Journal, 5(1): S. 61-96, 1998.

[DEC#93] Digital Equipment Corporation: *The DEC FUSE Handbook*. 1993.

[DeGr93] Deiters, W. und Gruhn, V.: *Software Process Validation Based on FUNSOFT Nets*. In: Proc. 2nd Europ. Workshop on Software Process Technology , Trondheim, Norway, Springer Verlag, LNCS 635, 1993, S. 50-52.

[DeHL96] Deiters, W., Herrmann, T. und Löffeler, T.: *Identifikation, Klassifikation und Unterstützung semistrukturierter Teilprozesse in prozeßorientierten Telekooperationssystemen*. In: Krcmar, H., Lewe, H. und Schwabe, G. (Hrsg.), Herausforderung Telekooperation - Einsatzerfahrungen und Lösungsansätze für ökonomische und ökologische, technische und soziale Fragen unserer Gesellschaft, DCSCW '96, Stuttgart-Hohenheim, Springer-Verlag Berlin, 1996, S. 261-274.

[DeKW99] Derniame, J.-C., Kaba, B.A. und Wastell, D. (Hrsg.): *Software Process: Principles, Methodology, and Technology*. Springer-Verlag, Berlin-Heidelberg, LNCS 1500, 1999.

[DeMa79] DeMarco, T.: *Structured Analysis and System Specification*. Prentice Hall, Englewood Cliffs, 1979.

[Demi86] Deming, W.E.: *Out of the Crisis*. Massachusetts Institute of Technology, Center for Advanced Engineering Study, Cambridge, USA, 1986.

[Dene93] Denert, E.: *Dokumentenorientierte Software-Entwicklung*. Informatik Spektrum, (16): S. 159-164, 1993.

[Dern94] Derniame, J.-C.: *Life Cycle Process Support in PCIS*. In: Proceedings of the PCTE'94 Conference, San Francisco, USA, 1994, S. 65-71.

[DGSZ94] Dinkhoff, G., Gruhn, V., Saalmann, A. und Zielonka, M.: *Business Process Modeling in the Workflow-Management Environment Leu*. In: Proceedings of the 13th International Conference on the Entity-Relationship Approach (ER '94), Manchester, UK, Springer Verlag, LNCS 881, 1994, S. 46-63.

[DHKL84] Donzeau-Gouge, V., Huet, G., Kahn, G. und Lang, B.: *Programming Environments Based on Structure Editors*: The Mentor Experience. In: Barstow, D.R., Shrobe, H.E. und Sandewall, E. (Hrsg.): Interactive Programming Environments. 1984, S. 128-1984.

[DHTT00] Damm, Ch. H., Hansen, K. M., Thomsen, M. und Tyrsted, M.: *Tool Integration: Experiences and Issues in Using XMI and Component Technology*. In: Proceedings of the TOOLS Europe 2000 Conference, Mont Saint-Michel/St-Malo, France, 2000, S. 94-107.

[DiGa00] Dittrich, K. und Gatziu, S.: *Aktive Datenbanken*. dpunkt Verlag, 2000.

[DoBe92] Dourish, P. und Bellotti, V.: *Awareness and Coordination in Shared Workspaces*. In: Proceeding of the Conference on Computer-Supported Cooperative Work (CSCW '92), Toronto, Canada, ACM Press, 1992, S. 107-114.

[DoD#80] US Department of Defense: *DoD Requirements for Ada Programming Support Environments*. "STONEMAN-Report", U.S. Department of Defense, High Order Language Working Group, AD-A100 404, 1980.

[DoD#88] US Department of Defense: *DoD-2176A Military Standard: Defense System Software Development*. U.S. Department of Defense, 1988.

[DoFe94] Dowson, M. und Fernström, Ch.: *Towards Requirements for Enactment Mechanisms*. In: Proc. 3rd Europ. Workshop on Software Process Technology, Villard de Lans, France, LNCS 772, Feb. 1994, S. 90-106.

[Döm*96] Dömges, R., Pohl, K., Jarke, M., Lohmann, B. und Marquardt, W.: *PRO-ART/CE - An Environment for Managing Chemical Process Simulation Models*. In: Proc. 10th European Simulation Multiconference, Budapest, Hungary, 1996, S. 1012-1016.

[Dömg99] Dömges, R.: *Projektspezifische Methoden zur Nachvollziehbarkeit von Anforderungsspezifikationen*. Dissertation, RWTH Aachen, 1999.

[Dono99] Donohoe, P. (Hrsg.): *Software Architecture - Proceedings of the 1st Working IFIP Conference on Software Architecture*. Kluwer Academic Publishing, Boston, 1999.

[DöPo98] Dömges, R. und Pohl, K.: *Adapting Traceability Environments to Project-Specific Needs*. Communications of the ACM, 41(12): S. 54-62, 1998.

[Dorl93] Dorling, A.: *SPICE: Software Process Improvement and Capacity dEtermination*. Information and Software Technology, 35(6/7): S. 404-406, 1993.

[Doug88] Douglas, J.: *Conceptual Design of Chemical Processes*. McGraw-Hill, 1988.

[Dows93] Dowson, M.: *Software Process Themes and Issues*. In: Proceedings of the 2nd International Conference on the Software Process, Berlin, Germany, IEEE Computer Society Press, 1993, S. 54-62.

[DrHM98] Dröschel, W., Heuser, W. und Midderhoff, R. (Hrsg.): *Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97*. Oldenbourg Verlag München Wien, 1998.

[DrHM98] Dröschel, W., Heuser, W. und Midderhoff, R. (Hrsg.): *Inkrementelle und objektorientierte Vorgehensweisen mit dem V-Modell 97*. Oldenbourg Verlag München Wien, 1998.

[EAMP97] Emmerich, W., Arlow, J., Madec, J. und Phoenix, M.: *Tool Construction for the British Airways SEE with the O2 ODBMS*. Theory and Practice of Object Systems, 3(3): S. 213-231, 1997.

[EBLA96] Emmerich, W., Bandinelli, S., Lavazza, L. und Arlow, J.: *Fine grained Process Modelling: An Experiment at British Airways*. In: Proc. 4th Intl. Conf. on the Software Process, 1996, S. 2-12.

[EbOO94] Eberleh, E., Oberquelle, H. und Oppermann, R. (Hrsg.): *Einführung in die Software-Ergonomie*. Walter de Gruyter, Berlin, New York, 1994.

[EbSt93] Ebenau, R.G. und Strauss, S.H.: *Software Inspection Process*. Mc Graw Hill, Systems Design & Implementation Series, 1993.

[EbSU97] Ebert, J., Süttenbach, R. und Uhe, I.: *Meta-CASE in Practice: a Case for KOGGE*. In: Olivé, A. und Pastor, J.A. (Hrsg.), Proc. 9th Intl. Conf. on Advanced Information System Engineering (CAiSE*97), Barcelona, Spain, Springer-Verlag, Berlin Heidelberg, LNCS 1250, 1997, S. 203-216.

[Ecke95] Eckerson, W.W.: *Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client/Server Applications*. Open Information Systems, 10(1): S. 3-23, 1995.

[ECMA93] ECMA: *Reference Model for Frameworks of Software Engineering Environments*. Tech. Report, European Computer Manufacturers Association (ECMA), TR/55 (3rd Edition), 1993.

[EdEd98] Eddon, G. und Eddon, H.: *Inside Distributed COM*. Microsoft Press, Redmond, Washington, 1998.

[EgKM00] Eggersmann, M., Krobb, C. und Marquardt, W.: *A Modeling Language for Design Processes in Chemical Engineering*. Erscheint in: Proceedings of the 19th International Conference on Conceptual Modeling (ER 2000), 2000.

[EIA#94] EIA: *The CDIF 1994 Interim Standard - Overview*. Tech. Report, Electronic Industries Association, EIA/IS-106, 1994.

[EiMC97] Eisenhauer, G., Mukherjee, B. und Codella, C.: *On the Implementation of CORBA Event Channels*. Tech. Report, T.J. Watson Reseach Center, RC 20947 (89322) 11AUG97, 1997.

[ElNa99] Elmasri, R. und Navathe, S.: *Fundamentals of Database Systems*. Addison-Wesley, 1999.

[EmFi96] Emmerich, W. und Finkelstein, A.: *Do Process-Centered Environments Deserve Process-Centered Tools?*. In: Montangero, C. (Hrsg.), Proc. of the 5th European Workshop on Software Process Technology, EWSPT '96, Springer Verlag, LNCS 1149, 1996, S. 75-81.

[Emme95] Emmerich, W.: *Tool Construction for Process-Centred Software Development Environments based on Object Databases*. Dissertation, University of Paderborn, Germany, 1995.

[Emme96] Emmerich, W.: *Tool Specification with GTSL*. In: Proc. 8th Intl. Workshop on Software Specification and Design, Schloß Velen, Germany, 1996, S. 26-35.

[Eng*92] Engels, G., Lewerentz, C., Nagl, M., Schäfer, W. und Schürr, A.: *Building Integrated Software Development Environments , Part I: Tool Specification*. ACM Transactions on Software Engineering and Methodology, 1(2): S. 135-167, 1992.

[Engl97] Englander, R.: *Developing Java Beans*. O'Reilly, 1997.

[ErPe98] Eriksson, H.-E. und Penker, M.: *UML Toolkit*. John Wiley & Sons, 1998.

[EsBa98] Estublier, J. und Barghouti, N.: *Interoperability and Distribution of Process-Sensitive Systems*. In: Proceedings of the Conference on Software Engineering for Parallel and Distributed Systems (PDSE '98), Kyoto, Japan, 1998, S. 103-115.

[EsCB98] Estublier, J., Cunin, P. und Belkhatir, N.: *Architectures for Process Support System Interoperability*. In: Proceedings of the 5th International Conference on the Software Process (ICSP 5), Chicago, Illinois, USA, 1998, S. 137-147.

[EsDa96] Estublier, J. und Dami, L.: *Process Engine Interoperability - An Experiment*. In: Proceedings of the 5th European Workshop on Software Process Technology (EWSPT 5), Nancy, France, Springer-Verlag, 1996, S. 43-60.

[Estu99] Estublier, J.: *Is a Process Formalism an Architecture Description Language?*. In: Proceedings of the International Process Technology Workshop (IPTW), Grenoble, Switzerland, 1999, S. 17-22.

[Faga76] Fagan, M.E.: *Design and Code Inspection to Reduce Errors in Program Development*. IBM Systems Journal, 15(3): S. 182-211, 1976.

[Faga86] Fagan, M.E.: *Advances in Software Inspections*. IEEE Transactions on Software Engineering, 12(7): S. 744-751, 1986.

[FaSJ99] Fayad, M., Schmidt, D.C. und Johnson, R. (Hrsg.): *Building Application Frameworks : Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 1999.

[FeHu93] Feiler, P. und Humphrey, W.S.: *Software Process Development and Enactment: Concepts and Definitions*. In: Proc. 2nd Intl. Conf. on Software Process, IEEE Computer Society Press, 1993, S. 28-39.

[FeNO92] Fernström, Ch., Närfelt, K.-H. und Ohlsson, L.: *Software Factory Principles, Architecture, and Experiments*. IEEE Software, S. 36-44, March 1992.

[FeOh91] Fernström, Ch. und Ohlsson, L.: *Integration Needs in Process Enacted Environments*. In: Procs. 2nd Intl. Conf. on the Software Process, 1991, S. 142-158.

[Fern93] Fernström, Ch.: *PROCESS WEAVER: Adding Process Support to UNIX*. In: Procs. 2nd Conf. on the Software Process, Berlin, Germany, Feb. 1993, S. 12-26.

[Fern93a] Fernström, Ch.: *State Models and Protocols in Process Centered Environments*. In: Proc. 8th Intl. Software Process Workshop: State of the Practice in Process Technology), Wadern, Germany, March 1993, S. 72-77.

[Fie*99] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. und Berners-Lee, T.: *Hypertext Transfer Protocol - HTTP/1.1* . IETF RFC 2616, 1999.

[FiKN94] Finkelstein, A., Kramer, J. und Nuseibeh, B. (Hrsg.): *Software Process Modelling and Technology*. RSP, John Wiley & Sons, London, 1994.

[FIPS93a] FIPS: *Integration Definition for Function Modeling (IDEF0)*. Tech. Report, Federal Information Processing Standards Publication, 183 (FIPS 183), 1993.

[FIPS93b] FIPS: *Integration Definition for Function Modeling (IDEF1X)*. Tech. Report, Federal Information Processing Standards Publication, 184 (FIPS 184), 1993.

[Flan00] Flanagan, D.: *Java in a Nutshell*. O'Reilly, 2000.

[FoKR94] Fowler, G., Korn, D. und Rao, H.: *n-DFS: The Multiple Dimensional File System*. In: Tichy, W. (Hrsg.), Trends in Software, Volume 2, John Wiley & Sons, Chichester, UK, 1994, S. 135-154.

[FoNo92] Forte, G. und Norman, R.J.: *A Self-Assessment by the Software Engineering Community*. Communications of the ACM, 35(4): S. 28-32, 1992.

[Fowl86] Fowler, G.: *In-Process Inspections of Workproducts at AT&T*. AT&T Technical Journal, 65(2): S. 102-112, 1986.

[FrAg96] Frolund, S. und Agha, G.: *Coordinating Distributed Objects*. MIT Press, Cambridge, MA, 1996.

[Fran91] Frankel, B.: *The ToolTalk Service*. Tech. Report, Sun Microsystems Inc., Mountain View, California, 1991.

[Frol93] Frolund, S.: *A Language for Multi-Object Coordination*. In: Proc. of 7th European Conference on Object-Oriented Programming (ECOOP '93), Springer-Verlag, Berlin, Heidelberg, LNCS 707, 1993, S. 346-360.

[FrWa93] Fromme, B. und Walker, J.: *An Open Architecture for Tool and Process Integration*. In: Proc. 6th Software Engineering Environments Conference (SEE '93), Reading, UK, IEEE CS Press, 1993, S. 50-62.

[FrWe82] Freedman, D.F. und Weinberg, G.M.: *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. Dorset House Publishing, New York, 1982.

[Fugg93] Fuggetta, A.: *A Classification of CASE Technology*. Computer, S. 25-38, Dec. 1993.

[Fugg96] Fuggetta, A.: *Functionality and Architecture of PSEEs*. Information and Software Technology, (38): S. 289-293, 1996.

[FuGh94] Fuggetta, A. und Ghezzi, C.: *State of the Art and Open Issues in Process-Centered Software Engineering Environments*. Journal of Systems and Software, 26(1): S. 53-60, July 1994.

[FuWo96] Fuggetta, A. und Wolf, A.L. (Hrsg.): *Trends in Software Process*. John Wiley & Sons, New York, Trends in Software, vol. 4, 1996.

[GaIl90] Garlan, D. und Ilias, E.: *Low-cost, Adaptable Tool Integration Policies for Integrated Environments*. S. 1-10, 1990.

[GaJa96] Garg, P. und Jazayeri, M. (Hrsg.): *Process-Centered Software Engineering Environments*. IEEE Computer Society Press, 1996.

[GaJa96a] Garg, P. und Jazayeri, M.: *Process-Centered Software Engineering Environments*. In: Fuggetta, A. und Wolf, A.L. (Hrsg.): Trends in Software Process. John Wiley & Sons, New York, Trends in Software, Vol. 4, 1996, S. 25-52.

[GaLK98] Gary, K., Lindquist, T. und Koehnemann, H.: *Component-based Software Process Support*. In: Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE '98), Honolulu, Hawaii, USA, Oct. 1998, S. 196-199.

[Gar*94] Garg, P., Mi, P., Pham, T., Scacchi, W. und Thunquest, G.: *The SMART Approach for Software Engineering Processes*. In: Proc. 16th International Conference on Software Engineering, Sorrento, Italy, IEEE CS Press, Los Alamitos, CA, 1994, S. 341-350.

[GaSa79] Gane, C. und Sarson, T.: *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, Englewood Cliffs, New Jersey, 1979.

[GeFi92] Genesereth, M. und Fikes, R.: *Knowledge Interchange Format v.3 Reference Manual*. Stanford University, 1992.

[GeHo94] Georgakopoulos, D. und Hornick, M.F.: *A Framework for Enforceable Specification of Extended Transaction Models and Transactional Workflows*. International Journal of Intelligent and Cooperative Information Systems, 3(3): S. 225-253, 1994.

[GHJV95] Gamma, E., Helm, R., Johnson, R.E. und Vlissides, J.: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GiKa91] Gisi, M.A. und Kaiser, G.E.: *Extending a Tool Integration Language*. In: Proc. 1st Intl. Conference on Software Process, Redondo Beach, CA, USA, Oct. 1991, S. 218-227.

[GoFi94] Gotel, O. und Finkelstein, A.: *An Analysis of the Requirements Traceability Problem*. In: Proceedings of the 1st International Conference on Requirements Engineering, Colorado Spring, Colorado, USA, IEEE Computer Society Press, 1994, S. 94-102.

[Gold84] Goldberg, A.: *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[GOOD94] GOODSTEP Team: *The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes*. In: Ohmaki, K. (Hrsg.), Proceedings of the Asia-Pacific Software Engineering Conference, Tokyo, Japan, IEEE Computer Society Press, 1994, S. 410-420.

[GrHM98] Grundy, J., Hosking, J. und Mugridge, W.B.: *Inconsistency Management for Multiple-View Software Development Environments*. IEEE Transactions on Software Engineering, 24(11): S. 960-981, Nov. 1998.

[Grif98] Griffel, F.: *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt Verlag Heidelberg, 1998.

[Gro*97] Grosz, G., Rolland, C., Schwer, S., Souveyet, C., Si-Said, S., Ben Achour, C. und Gnaho, C.: *Modelling and Engineering the Requirements Engineering Process: An Overview of the NATURE Approach*. Requirements Engineering Journal, No. 2: S. 115-131, 1997.

[GrRW00] Greenwood, R.M., Robertson, I. und Warboys, B.: *A Support Framework for Dynamic Organizations*. In: Proceedings of the 7th European Workshop on Software Process Technology, Kaprun, Österreich, Springer-Verlag, LNCS 1780, 2000

[Grun99] Grundmann, N.: *Realisierung und Visualisierung von UML-Modellen in Telos*. Diplomarbeit, RWTH Aachen, 1999.

[Gry*99] Gryczan, G., Lilienthal, C., Lippert, M., Roock, S., Wolf, H. und Züllighoven, H.: *Frameworkbasierte Anwendungsentwicklung (Teil 1)*. OBJEKTspektrum, 1/99: S. 90-99, 1999.

[HaHK97] Harsu, M., Hautamäki, J. und Koskimies, K.: *A Language Implementation Framework in Java*. In: Proc. of the European Conference on Object-Oriented Technology, ECOOP 97, Springer-Verlag, LNCS 1357, 1997.

[HaLe93] Habermann, H.J. und Leymann, F. (Hrsg.): *Repository - Eine Einführung*. Oldenbourg, München, 1993.

[HaNo86] Habermann, N. und Notkin, D.: Gandalf: *Software Development Environments*. IEEE Transactions on Software Engineering, 12(12): S. 1117-1127, 1986.

[HaPW98] Haumer, P., Pohl, K. und Weidenhaupt, K.: *Requirements Elicitation and Validation with Real World Scenes*. IEEE Transactions on Software Engineering, 24(12): S. 1036-1054, 12 1998.

[Hare87] Harel, D.: *Statecharts: a visual formalism for complex systems*. Science of Computer Programming, 8: S. 231-274, 1987.

[Harm97] Harmsen, F.: *Situational Method Engineering*. Dissertation, Moret Ernst & Young, Utrecht, NL, 1997.

[HaSa96] Harmsen, F. und Saeki, M.: *Comparison of four Method Engineering Languages*. In: Brinkkemper, S., Lyytinen, K. und Welke, R. (Hrsg.): Method Engineering - Principles of Method Construction. IFIP Chapman & Hall, 1996, S. 209-231.

[Haum00] Haumer, P.: *Requirements Engineering with Interrelated Conceptual Models and Real-World Scenes*. Dissertation, RWTH Aachen, 2000.

[Hay*00] Hayes, J.G., Peyrovian, E., Sarin, S., Schmidt, M.-T., Swenson, K.D. und Weber, R.: *Workflow Interoperability Standards for the Internet*. IEEE Internet Computing, S. 37-45, May/June 2000.

[HeEd93] Henderson-Sellers, B. und Edwards, J.M.: *Object-Oriented Knowledge: The Working Object*. Prentice-Hall, 1993.

[Heim90] Heimbigner, D.: *Proscription versus Prescription in Process Centered Environments*. In: Proc. of the 6th Intl. Software Process Workshop: Support for the Software Process, Hakodate, Japan, Oct. 1990, S. 99-102.

[Heim92] Heimbigner, D.: *The ProcessWall: A Process State Server Approach to Process Programming*. In: Procs. 5th ACM SIGSOFT/SIGPLAN Symposium on Software Development Environments, Dec. 1992, S. 159-168.

[HeKW97] Heimann, P., Krapp, C.-A. und Westfechtel, B.: *An Environment for Managing Software Development Processes*. In: Proc. 8th Conference on Software Engineering Environments, Cottbus, Germany, April 1997, S. 101-109.

[Herc94] Herczeg, M.: *Software-Ergonomie*. Addison-Wesley, 1994.

[HiKa99] Hitz, M. und Kappel, G.: *UML@Work*. dpunkt Verlag, 1999.

[HJKW96] Heimann, P., Joeris, G., Krapp, C.-A. und Westfechtel, B.: *DYNAMITE: Dynamic Task Nets for Software Process Management*. In: Proc. 18th Intl. Conf. on Software Engineering, Berlin, Germany, 1996, S. 331-341.

[Hoar85] Hoare, C.A.R.: *Communication Sequential Processes*. Printice Hall, Englewood Cliffs, New Jersey, 1985.

[Holl95] Hollingsworth, D.: *The Workflow Reference Model*. Tech. Report, Workflow Management Coalition, TC00-1003, 1995.

[Hor*98] Horvitz, E., Bresse, J., Heckerman, D., Hovel, D. und Rommelse, K.: *The Lumière Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users*. In: Proc. 14th Conference on Uncertainty in Artificial Intelligence, Madison, WI, 1998, S. 256-265.

[HoVe97] ter Hofstede, A.H.M. und Verhoef, T.F.: *On the Feasibility of Situational Method Engineering*. Information Systems, 22(6/7): S. 401-422, 1997.

[Huff96] Huff, K.E.: *Software Process Modeling*. In: Fuggetta, A. und Wolf, A.L. (Hrsg.): Trends in Software Process. John Wiley & Sons, New York, Trends in Software, Vol. 4, 1996, S. 1-24.

[Hump89] Humphrey, W.S.: *Managing the Software Process*. Addison-Wesley, 1989.

[IBM#84] IBM: *Business Systems Planning - Information Systems Planning Guide*. Application Manual, IBM Corporation, 1984.

[IEEE95] IEEE: *IEEE Standard for Developing Software Life Cycle Processes*. Tech. Report, IEEE 1074, 1995.

[Iiva96] Iivari, J.: *Why are CASE Tools not used ?*. Communications of the ACM, 39(10): S. 94-103, 1996.

[Iona00] Iona Technologies: *OrbixCOMet*. 2000. (http://www.iona.com/products/orbix/comet.html)

[IRDS90] IRDS: *Information Technology - Information Resource Dictionary System (IRDS) Framework*. Tech. Report, ISO/IEC, ISO/IEC 10027, 1990.

[ISO#91] ISO: *ISO 9000-3: Quality Management and Quality Assurance Standards*. International Organization for Standardization, Genf, Schweiz, 1991.

[ISO#94] ISO: *Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 11: Description Methods: The EXPRESS Language Reference Manual*. International Organisation for Standardization, ISO 10303-11, 1994.

[ISO#95] ISO/IEC: *International Standard Information Technology Software Life Cycle Process*. Tech. Report, ISO 12207, 1995

[JaBR99] Jacobson, I., Booch, G. und Rumbaugh, J.: *The Unified Software Development Process*. Addison Wesley, 1999.

[JaBu96] Jablonski, St. und Bussler, Ch.: *Workflow Management - Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.

[JaDB89] Jagannathan, V., Dodhiawala, R. und Baum, R.S. (Hrsg.): *Blackboard Architectures and Applications*. Academic Press, New York, 1989.

[JaJa95] Jacobson, I. und Jacobson, St.: *Beyond Methods and CASE: the Software Engineering Process with its Integral Support Environment*. Object Magazine, Jan. 1995.

[JaHu98] Jarzabek, St. und Huang, R.: *The Case for User-Centered CASE Tools*. Communications of the ACM, 41(8): S. 93-99, Aug. 1998.

[JaJQ99] Jarke, M., Jeusfeld, M.A., Quix, Ch. (Hrsg.): *ConceptBase V5.1 User Manual*. Tech. Report, RWTH Aachen, 1999.

[JaLW99] Jarke, M., List, Th. und Weidenhaupt, K.: *A Process-Integrated Conceptual Design Environment for Chemical Engineering*. In: Proc. 18th Intl. Conf. on Conceptual Modeling (ER '99), Paris, Frankreich, Springer Verlag, Nov. 1999, S. 520-537.

[JaMa96] Jarke, M. und Marquardt, W.: *Design and Evaluation of Computer-Aided Process-Modeling Tools*. In: Davis, J.F., Stephanopoulos, G. und Venkatasubramaniam, V. (Hrsg.): Proc. Intl. Conf. on Intelligent Systems in Process Engineering. AIChE Symposium Series Vol. 92, No. 312, 1996, S. 97-109.

[Jann92] Janning, Th.: *Requirements Engineering und Programmierung im Großen - Integration von Sprachen und Werkzeugen*. Dissertation, RWTH Aachen, Deutscher Universitätsverlag, Wiesbaden, 1992.

[Jar*95] Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M. und Eherer, St.: *ConceptBase - A Deductive Object Base for Meta Data Management*. Journal of Intelligent Information Systems, 4(2): S. 167-192, 1995.

[Jar*98] Jarke, M., Pohl, K., Weidenhaupt, K., Lyytinen, K., Marttiin, P., Tolvanen, J.-P. und Papazoglou, M.: *Meta Modelling: A Formal Basis for Interoperability and Adaptability*. In: Krämer, B., Papazoglou, M. und Schmidt, H.-W. (Hrsg.): Information Systems Interoperability. Research Studies Press, Taunton, Somerset, England, 1998, S. 229-263.

[Jar*98a] Jarke, M., List, Th., Nissen, H.W., Lohmann, B. und Hubbuch, K.: *Bericht zum Workshop "Verfahrenstechnische Datenbanken"*. Interner Bericht, Bayer AG, 1998.

[Java98] JavaSoft: *The JavaBeans Bridge for ActiveX*. Feb. 1998. (http://java.sun.com/products/javabeans/software/bridge/)

[JCJÖ92] Jacobson, I., Christerson, M., Jonsson, P. und Övergaard, G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

[JePa97] Jeusfeld, M.A. und Papazoglou, M.: *Information Brokering*. In: Krämer, B., Papazoglou, M. und Schmidt, H.-W. (Hrsg.), Information Systems Interoperability, Research Studies Press, 1997, S. 265-302.

[Jeus92] Jeusfeld, M.A.: *Änderungskontrolle in deduktiven Objektbanken*. Dissertation, Universität Passau, 1992.

[JPRS94] Jarke, M., Pohl, K., Rolland, C. und Schmitt, J.-R.: *Experience-Based Method Evaluation and Improvement: A Process Modeling Approach.* In: Proceedings of the IFIP 8.1 CRIS94 Working Conference: Methods and Associated Tools for the Informations Systems Life Cycle, Maastricht, Netherlands, 1994, S. 1-27.

[JPSW94] Junkermann, G., Peuschel, B., Schäfer, W. und Wolf, St.: *MERLIN: Supporting Cooperation in Software Development through a Knowledge-Based Environment.* In: Finkelstein, A., Kramer, J. und Nuseibeh, B. (Hrsg.): Software Process Modelling and Technology. Research Studies Press, 1994, S. 103-130.

[JRSD99] Jarke, M., Rolland, C., Sutcliffe, A. und Dömges, R. (Hrsg.): *The NATURE of Requirements Engineering.* Shaker-Verlag, Aachen, 1999.

[KaRo74] Kast, F.E. und Rosenzweig, J.E.: *Organization and Management: A Systems Approach.* McGraw-Hill, 1974.

[Katz90] Katz, R.H.: *Towards a Unified Framework for Version Modeling in Engineering Databases.* ACM Computing Surveys, 22(4): S. 375-408, 1990.

[Kel*98] Kellner, M., Becker-Kornstaedt, U., Riddle, W., Tomal, J. und Verlage, M.: *Process Guides: Effective Guidance for Process Participants.* In: Proc. 5th Intl. Conf. on the Software Process: Computer Supported Organizational Work, Lisle, Illinois, USA, June 14-17 1998, S. 11-25.

[KeLR95] Kelly, St., Lyytinen, K. und Rossi, M.: MetaEdit+: *A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment.* In: Procs. 7th Intl. Conf. on Advanced Information Systems Engineering, CAiSE '95, Jyväskylä, Finland, June 1995, S. 1-21.

[Kelt93] Kelter, U.: *Integrationsrahmen für Software-Entwicklungsumgebungen.* Informatik Spektrum, (16): S. 281-285, 1993.

[KeMo93] Kemper, A. und Moerkotte, G.: *Basiskonzepte objektorientierter Datenbanksysteme.* Informatik Spektrum, (16): S. 69-80, 1993.

[KeRi84] Kernighan, B.W. und Ritchie, R.P.: *The UNIX Programming Environment.* Prentice Hall, Englewood Cliffs, 1984.

[KeSm96] Kelly, St. und Smolander, K.: *Evolution and Issues in metaCASE.* Information and Software Technology, (38): S. 261-266, 1996.

[Kipe94] Kiper, J.D.: *A Framework for Characterisation of the Degree of Integration of Software Tools.* Journal of Systems Integration, Vol. 4: S. 5-32, 1994.

[KiSW95] Kiesel, N., Schuerr, A. und Westfechtel, B.: GRAS, A Graph-Oriented (Software) Engineering Database System. Information Systems, 20(1): S. 21-51, 1995.

[KiRB91] Kiczales, G., des Rivières, J. und Bobrow, D.G.: *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Klam95] Klamma, R.: *Präskriptive Prozessbeschreibungen: Definition, Implementierung und Validierung in PRO-ART.* Diplomarbeit, RWTH Aachen, 1995.

[Kobr99] Kobryn, C.: *UML 2001: A Standardization Odyssey.* Communications of the ACM, 42(10): S. 29-37, Oct. 1999.

[Koch93] Koch, G.R.: *Process Assessment: The "BOOTSTRAP" approach.* Information and Software Technology, 35(6/7): S. 387-403, 1993.

[KoKo84] Kotteman, J. und Konsynsky, B.: *Information Systems Planning and Development: Strategic Postures and Methodologies.* Journal of Management Information Systems, 1(2): S. 45-63, 1984.

[KoMö95] Koskimies, K. und Mössenböck, H.: *Designing a Framework by Stepwise Generalization.* In: Proc. of the 5th European Software Engineering Conference, Springer-Verlag, LNCS989, 1995.

[Kräm97] Krämer, B.: *Distributed Object Platforms: The CORBA Standard.* In: Krämer, B., Papazoglou, M. und Schmidt, H.-W. (Hrsg.): Information Systems Interoperability. Research Press Studies, Taunton, Somerset, England, 1997, S. 13-38.

[KrMa97] Kramer, J. und Magee, J.: *Exposing the Skeleton in the Coordination Closet.* In: Garlan, D. und Métayer, D.L. (Hrsg.), Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION '97), Berlin, Germany, 1997, S. 18-31.

[Krob97] Krobb, C.: *Entwicklung einer Spezialisierungshierarchie für Modellierungsschritte im objektorientierten Datenmodell VeDa*. Diplomarbeit, RWTH Aachen, 1997.

[Kron92] Kronlöf, K. (Hrsg.): *Method Integration: Concepts and Case Studies*. John Wiley & Sons, 1992.

[KrRo95] Krishnamurthy, B. und Rosenblum, D.S.: *Yeast: A General Purpose Event-Action System*. IEEE Transactions on Software Engineering, 21(10): S. 845-857, Oct. 1995.

[Kruc98] Kruchten, Ph.: *The Rational Unified Process - An Introduction*. Addison Wesley, 1998.

[KuWe92] Kumar, K. und Welke, R.J.: *Methodology Engineering: A Proposal for Situation-specific Methodology Engineering*. In: Cotterman, W.W. und Senn, J.A. (Hrsg.): Challenges and Strategies for Research in Systems Development. John Wiley & Sons, Chichester, UK, 1992, S. 257-269.

[LaBo97] Lang, K. und Bodendorf, F.: *Gestaltung von Geschäftsprozessen auf der Basis von Prozeßbausteinbibliotheken*. In: Business Process (Re-)Engineering, dpunkt-Verlag, Nr. 198, Theorie und Praxis der Wirtschaftsinformatik/Handbuch der maschinellen Datenverarbeitung, 1997, S. 83-93.

[Lamp99] Lamprecht, St.: *Programmieren für das WWW mit JavaScript, VBScript, XML und SMIL*. Verlag Carl Hanser, 1999.

[Laur90] Laurel, B.: *Interface Agents: Metaphors with Character*. In: Laurel, B. (Hrsg.): The Art of Human Computer Interface Design. Addison-Wesley, Band 1, 1990, S. 355-365.

[LaVo97] Lausen, G. und Vossen, G.: *Models and Languages of Object-Oriented Databases*. Addison Wesley , International Computer Science Series, 1997.

[Lefe95] Lefering, M.: *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*. Dissertation, RWTH Aachen, Verlag Shaker, 1995.

[Lehm87] Lehmann, M.M.: *Process Models, Process Programs, Programming Support*. In: Proc. 9th Intl. Conf. on Software Engineering, Monterey, California, USA, 1987, S. 14-16.

[Lehm91] Lehmann, M.M.: *Software Engineering, the Software Process and Their Support*. Software Engineering Journal, Sept. 1991.

[LeYo94] Lee, J. und Yost, G.: *The PIF Process Interchange Format and Framework*. Tech. Report, PIF Working Group, 1994.

[LiKS99] Ließmann, H., Kaufmann, Th. und Schmitzer, B.: *Bussysteme als Schlüssel zur betriebswirtschaftlich-semantischen Kopplung von Anwendungssystemen*. Wirtschaftsinformatik, 41(1): S. 12-19, Jan. 1999.

[Lohm98] Lohmann, B.: *Verfahrenstechnische Modellierungsabläufe*. Dissertation, RWTH Aachen, VDI Verlag Düsseldorf, Fortschritts-Berichte VDI, Reihe 3, Nr. 531, 1998.

[Lonc94a] Lonchamp, J.: *An Assessment Exercise*. In: Finkelstein, A., Kramer, J. und Nuseibeh, B. (Hrsg.): Software Process Modelling and Technology. RSP by John Wiley & Sons, 1994, S. 335-356.

[Lott93] Lott, Ch.: *Process and Measurement Support in SEEs*. ACM SIGSOFT Software Engineering Notes, 18(4): S. 83-93, 1993.

[Luc*95] Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D. und Mann, W.: *Specification and Analysis of System Architecture using Rapide*. IEEE Transactions on Software Engineering, 21(4): S. 336-355, 1995.

[LyWe99] Lyytinen, K. und Welke, R. (Hrsg.): *Special Issue on Meta-Modelling and Methodology Engineering*. Information Systems, 24(2): S. 67-69, 1999.

[Lyy*98] Lyytinen, K., Marttiin, P., Tolvanen, J.-P., Jarke, M., Pohl, K. und Weidenhaupt, K.: *CASE Environment Adaptability: Bridging the Islands of Automation*. In: March, S.T. und Bubenko, J. (Hrsg.), Proceedings of the 8th Annual Workshop on Information Technologies and Systems (WITS '98), 1998, S. 115-125.

[MaBF99] Mattsson, M., Bosch, J. und Fayad, M.: *Framework Integration - Problems, Causes, Solutions*. Communications of the ACM, 42(10): S. 81-87, Oct. 1999.

[MaCr94] Malone, T. und Crowston, K.: *The Interdisciplinary Study of Coordination*. ACM Computing Surveys, 26(1): S. 87-119, 1994.

[Madh91] Madhavji, N.H.: *The Process Cycle*. Software Engineering Journal, 6(5): S. 234-242, 1991.

[Maes94] Maes, P.: *Agents that Reduce Work and Information Overload.* Communications of the ACM, 37(7): S. 30-40, 1994.

[MaFS97] Mark, G., Fuchs, L. und Sohlenkamp, M.: *Supporting Groupware Conventions through Contextual Awareness.* In: Prinz, W., Rodden, T., Hughes, H. und Schmidt, K. (Hrsg.), Proceedings of the Fifth European Conference on Computer Supported Cooperative Work (ECSCW'97), Lancaster, UK, Kluwer Academic Publishers, 1997, S. 253-268.

[Mart98] Marttiin, P.: *Customisable Process Modeling Support and Tools for Design Environment.* Dissertation, University of Jyväskylä, 1998.

[Maz*94] Mazza, C., Fairclough, J., Melton, B., Pablo, D.D., Scheffer, A. und Stevens, R.: *Software Engineering Standards.* 1994.

[MBJK90] Mylopoulos, J., Borgida, A., Jarke, M. und Koubarakis, M.: *Telos - Representing Knowledge about Information Systems.* ACM Transactions on Information Systems, 8(4): S. 325-362, 1990.

[McCh95] McChesney, I.R.: *Towards a Clssification Scheme for Software Process Modeling Approaches.* Information and Software Technology, 37(7): S. 363-374, 1995.

[McPa84] McMenamin, S.M. und Palmer, J.F.: *Essential System Analysis.* Yourdon Press, Prentice Hall, Englewood Cliffs, 1984.

[MDEK95] Magee, J., Dulay, N., Eisenbach, S. und Kramer, J.: *Specifying Distributed Software Architectures.* In: Schäfer, W. und Botella, B. (Hrsg.), Proc. 5th European Software Engineering Conference, Barcelona, Spain, Springer-Verlag, LNCS 989, 1995, S. 137-153.

[MDKW99] Montangero, C., Derniame, J.-C., Kaba, B.A. und Warboys, B.: *The Software Process: Modelling and Technology.* In: Derniame, J.-C., Kaba, B.A. und Wastell, D. (Hrsg.): Software Process: Principles, Methodology, and Technology. Springer-Verlag Heidelberg Berlin, LNCS 1500, 1999, S. 1-14.

[MeTa97] Medvidovic, N. und Taylor, R.: *A Framework for Classifying and Comparing Architecture Description Languages.* In: Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '97), Zürich, Switzerland, 1997, S. 60-76.

[Meye97] Meyer, B.: *Object-Oriented Software Construction.* Prentice Hall, 1997.

[Meye90] Meyer, B.: *Introduction to a Theory of Programming Languages.* Englewodd Cliffs, NJ, 1990.

[Meye91] Meyers, S.: *Difficulties in Integrating Multiview Development Systems.* IEEE Software, S. 49-57, Jan. 1991.

[Micr95] Microsoft: *The Windows Interface Guidelines for Software Design.* Microsoft Press, Richmond, 1995.

[Micr97] Microsoft Corp.: *IntelliSense in Microsoft Office 97.* Tech. Report, Microsoft Office White Paper, 1997.

[Miln89] Milner, R.: *Communication and Currency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Mint97] Mintert, S.: *JavaScript 1.2: Einführung, Referenz, Praxislösungen.* Addison-Wesley, 1997.

[MiSc92] Mi, P. und Scacchi, W.: *Process Integration in CASE Environments.* IEEE Software, S. 45-53, March 1992.

[Mont94] Montangero, C.: *The Process in the Tool Syndrome: is it becoming worse?.* In: Proc. of the 9th Intl. Software Process Workshop, Arlie, VA, USA, 10 1994, S. 53-56.

[MüSc96] Müller, O. und Scholz, P.: *Specification of Real-Time and Hybrid Systems in FOCUS.* Tech. Report, Technische Universität München, TUM-I9627, 1996.

[MyCN92] Mylopoulos, J., Chung, L. und Nixon, B.: *Representing and Using Non-Functional Requirements: A Process-Oriented Approach.* IEEE Transactions on Software Engineering, 18(6): S. 483-497, 1992.

[Nagl90] Nagl, M.: *Softwaretechnik - Methodisches Programmieren im Großen.* Springer-Verlag, 1990.

[Nagl96] Nagl, M. (Hrsg.): *Building Tightly Integrated Software Development Environments: The IPSEN Approach.* Springer-Verlag, LNCS 1170, 1996.

[NATU96] NATURE Team: *Defining Visions in Context: Models, Processes and Tools for Requirements Engineering.* Information Systems, 21(6): S. 515-547, 1996.

[NaWe99] Nagl, M. und Westfechtel, B. (Hrsg.): *Integration von Entwicklungssystemen in Ingenieuranwendungen - Substantielle Verbesserung der Entwicklungsprozesse.* Springer, 1999.

[NiDa95] Nierstrasz, O. und Dami, L.: *Component-Oriented Software Technology.* In: Nierstrasz, O. und Tsichritzis, D. (Hrsg.), Object-Oriented Software Composition, Prentice-Hall, London, 1995, S. 3-28.

[NiJa99] Nissen, H.W. und Jarke, M.: *Repository Support for Multi-Perspective Requirements Engineering.* Information Systems (Special Issue on Meta Modeling an Method Engineering), 24(2): S. 131-158, 1999.

[Nils89] Nilsson, E.G.: *CASE Tools and Software factories.* In: Steinholtz, B., Solvberg, A. und Bergmann, L. (Hrsg.), Proc. CASE'89 (First Nordic Conference on Advanced Systems Engineering), Stockholm, Sweden, May 1989, S. 42-60.

[NIST93] NIST: *A Reference Model for Project Support Environment Standards.* Tech. Report, SEI & NIST, CMU/SEI-TR-93-23, NIST Report SP 500-213, Nov. 1993.

[Ober96] Oberweis, A.: *Modellierung und Ausführung von Workflows mit Petri-Netzen.* Teubner-Verlag, Stuttgart, 1996.

[Odel96] Odell, J.: *A Primer to Method Engineering.* In: Brinkkemper, S., Lyytinen, K. und Welke, R. (Hrsg.): Method Engineering - Principles of Method Construction. IFIP Chapman & Hall, 1996, S. 1-7.

[Oll*91] Olle, T.W., Hagelstein, J., Macdonald, I.G., Rolland, C., Sol, H.G., Van Assche, F. und Verrijn-Stuart, A.A.: *Information Systems Methodologies - A Framework for Understanding.* Addison Wesley, 1991.

[OMG#97] OMG: *The Common Object Request Broker: Architecture and Specification Revision 2.0.* Tech. Report, Object Management Group, OMG Document 97-07-04, 1997.

[OMG#97a] OMG: *What Is OMG-UML and Why Is It Important?.* Tech. Report, Object Management Group, OMG's Press Releases 1997, 1997. (http://www.omg.org/news/pr97/umlprimer.html)

[OMG#97b] OMG: *CORBAservices: Common Object Services Specification.* Tech. Report, Object Management Group, July 1997. (ftp://ftp.omg.org/pub/docs/formal/97-07-04.pdf)

[OMG#97c] OMG: *Meta Object Facility (MOF) Specification.* Tech. Report, Object Management Group, OMG Document ad/97-08-14, September 1997.

[OMG#97d] OMG: *Object Constraint Language Specification.* Object Management Group, OMG Document ad/97-08-08, September 1997.

[OMG#98a] OMG: Workflow Management Facility (OMG BODTF RFP #2 Submission). Tech. Report, OMG, OMG Document Number: bom/98-06-07, July 1998.

[OrHE96] Orfali, R., Harkey, D. und Edwards, J.: *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, 1996.

[Ortn99] Ortner, E.: *Repository Systems. Teil 1: Mehrstufigkeit und Entwicklungsumgebung.* Informatik Spektrum, 22(4): S. 235-251, 1999.

[Oust94] Ousterhout, J.K.: *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Park92] Parker, B.: *Introducing EIA-CDIF: The CASE Data Interchange Format Standard.* In: Proc. 2nd Symposium on the Assessment of Quality Software Development Tools, New Orleans, LA, USA, May 1992.

[PaSa96] Paul, G. und Sattler, K.-U.: *MediatorService - Integration von verteilten Objekten durch Beschreibung der Interaktionen.* In: Spaniol, O., Linnhoff-Popien, C. und Meyer, B. (Hrsg.), Proceedings of the International Workshop "Trends in Distributed Systems", Aachen, Germany, 1996, S. 125-132.

[PaSE97] Paul, G., Sattler, K.-U. und Endig, M.: *An Integration Framework for Open Tool Environments.* In: Distributed Applications and Interoperable Systems, Proceedings of International Working Conference (IFIP WG 6.1), Cottbus, Germany, 1997, S. 193-200.

[PCCW93] Paulk, M.C., Curtis, B., Chrissis, M.B. und Weber, C.V.: *Capability Maturity Model: Version 1.1.* IEEE Software, 10(4): S. 18-27, 1993.

[Pint93] Pintado, X.: *Gluons: a Support for Software Component Cooperation.* In: Nishio, S. und Yonezama, A. (Hrsg.), Proc. of the International Symposium on Object Technolo-

gies for Advanced Software (ISOTAS '93), Springer-Verlag, Berlin, Heidelberg, LNCS 742, 1993, S. 43-60.

[PlRo95] Plihon, V. und Rolland, C.: *Modelling Ways-of-Workings*. In: Proceedings of the 7th International Conference on Advanced Information Systems Engineering (CAiSE '95), Jyväskylä, Finnland, 1995, S. 126-139.

[Poh*99] Pohl, K., Weidenhaupt, K., Dömges, R., Haumer, P., Jarke, M. und Klamma, R.: *PRIME: Towards Process-Integrated Environments*. ACM Transactions on Software Engineering and Methodology, 8(4): S. 343-410, 1999.

[PoHa95] Pohl, K. und Haumer, P.: *HYDRA: A Hypertext Model for Structuring Informal Requirements Representations*. In: Proceedings of the 2nd International Workshop on Requrements Engineering: Foundation of Software Quality, Jyväskylä, Finnland, 1995, S. 118-134.

[Pohl94] Pohl, K.: *The Three Dimensions of Requirements Engineering: A Framework and Its Applications*. Information Systems, 19(3): S. 243-258, 1994.

[Pohl95] Pohl, K.: *A Process-Centered Requirements Engineering Environment*. Dissertation, RWTH Aachen, 1995.

[Pohl96] Pohl, K.: *Process-Centered Requirements Engineering*. Research Studies Press, 1996.

[Pohl97] Pohl, K.: *Let's Put the Stakeholders Performing the Process in the Driver's Seat*. In: Proceedings of the International Workshop on Research Directions in Process Technology, Nancy, Frankreich, 1997

[Pohl99] Pohl, K.: *Continuous Documentation of Information Systems Requirements*. Habilitation, RWTH Aachen, 1999.

[PoSW96] Popien, C., Schürmann, G. und Weiß, K.: *Verteilte Verarbeitung in Offenen Systemen*. B.G. Teubner-Verlagsgesellschaft, Stuttgart, 1996.

[PoWe97] Pohl, K. und Weidenhaupt, K.: *A Contextual Approach for Process-Integrated Tools*. In: Jazayeri, M. und Schauer, H. (Hrsg.), Proceedings of the 6th European Software Engineering Conference (ESEC/FSE '97), Zurich, Switzerland, Springer-Verlag, LNCS 1301, 1997, S. 176-192.

[Pree97a] Pree, W.: *Essential Framework Design Patterns*. Object Magazine, 7(1), March 1997.

[Pree97b] Pree, W.: *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt.verlag, 1997.

[Pres97] Pressman, R.S.: *Software Engineering*. In: Thayer, R.H. (Hrsg.): Software Engineering - Project Management. IEEE Computer Society Press, 1997, S. 30-47.

[Purt94] Purtilo, J.M.: *The Polylith Software Bus*. ACM Transactions on Programming Languages and Systems, 16(1): S. 151-174, 1994.

[PuTV97] Puustjärvi, J., Tirry, H. und Veijalainen, J.: *Reusability and Modularity in Transactional Workflows*. Information Systems, 22(2/3): S. 101-120, 1997.

[RaSt92] Rammig, F.J. und Steinmüller, B.: *Frameworks und Entwurfsumgebungen*. Informatik Spektrum, 15(1): S. 33-43, 1992.

[Redw93] Redwine, S.T.: *Humans and Processes - IWSP8 Session Summary*. In: Proc. of the 8th Intl. Software Process Workshop, Wadern , Germany, IEEE Computer Society Press, 1993, S. 12-14.

[Reif93] Reifer, D.J.: *Managing the Three P's: The Key Success to Software Management*. In: Reifer, D.J. (Hrsg.): Software Management. IEEE Computer Society Press, 1993, S. 2-8.

[Reis90] Reiss, St.: *Connecting Tools Unsing Message Passing in the Field Environment*. IEEE Software, 7(4): S. 57-66, July 1990.

[Reis90a] Reiss, St.: *Interacting with the Field Environment*. Software Practice and Experience, 20(S1): S. 89-115, Juni 1990.

[Reit98] Reiter, Ch.: *Toolbasierte Referenzmodellierung - State-of-the-Art und Entwicklungstrends*. In: Becker, J., Rosemann, M. und Schütte, R. (Hrsg.): Referenzmodellierung: State-of-the-Art and Entwicklungsperspektiven. Physica-Verlag, 1998, S. 45-68.

[ReTe81] Reps, T. und Teitelbaum, T.: *The Cornell Program Synthesizer: A Syntax-directed Programming Environment*. Communications of the ACM, 24(9): S. 449-477, 1981.

[ReTe88] Reps, T. und Teitelbaum, T.: *The Synthesizer Generator - A System for Constructing Language Based Editors*. Springer, New York, 1988.

[Roll97] Rolland, C.: *A Primer for Method Engineering.* In: Proc. of the INFORSID Conference (INformatique des ORganisations et Systèmes d'Information et de Décision), Toulouse, France, 1997.

[RoPB99] Rolland, C., Prakash, N. und Benjamen, A.: *A Multi-Model View of Process Modelling.* Requirements Engineering, 4(4): S. 169-187, 1999.

[RoSB98] Rolland, C., Souveyet, C. und Ben Achour, C.: *Guiding Goul Modelling using Scenarios.* IEEE Transactions on Software Engineering, Special Issue on Scenario Management, 24(12): S. 1055-1071, 1998.

[RoSc77] Ross, T.R. und Schoman, K.E.: *Structured Analysis for Requirements Definition.* IEEE Transactions on Software Engineering, 3(1): S. 6-15, 1977.

[RoSc99] Rosenberg, D. und Scott, K.: *Use Case Driven Object Modeling with UML.* 1999.

[RoSM95] Rolland, C., Souveyet, C. und Moreno, M.: *An Approach for Defining Ways-of-Working.* Information Systems, 20(4): S. 337-359, 1995.

[Roth93] Roth, Ch.: *Die Auswirkungen von CASE.* In: Goerke, W. und Rininsland, H. (Hrsg.), Information als Produktionsfaktor, Springer Verlag, Informatik aktuell, 1993, S. 648-656.

[Royc70] Royce, W.W.: *Managing the Development of Large Software Systems.* In: Procs. Wescon, New York, USA, IEEE Computer Society Press, (Nachgedruckt in: Proc. 9th Intl. Conf. on Software Engineering, 1987, S. 328-338), 1970, S. 1-9.

[RuJB99] Rumbaugh, J., Jacobson, I. und Booch, G.: *The Unified Modeling Language Reference Manual.* Addison Wesley, 1999.

[Rum*91] Rumbaugh, J., Blaha, M.R., Premerlani, W.J., Eddy, F. und Lorensen, W.: *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[Sarl92] Sarlan, H.: *Managementaspekte bei objektorientierten Entwicklungsprojekten.* Informatik Spektrum, 15(5): S. 282-286, 1992.

[Satt97] Sattler, K.-U.: *A Framework for Component-Oriented Tool Integration.* In: Proceedings of the 4th International Conference on Object-Oriented Information Systems (OOIS'97), Brisbane, Australia, 1997, S. 455-465.

[ScBr93] Schefström, D. und Broek, G.van den (Hrsg.): *Tool Integration - Environments and Frameworks.* John Wiley & Sons, 1993.

[Scha96] Schach, S.R.: *Classical and Object-Oriented Software Engineering.* IRWIN, 1996.

[Sche93] Schefström, D.: *System Development Environments: Contemporary Concepts.* In: Schefström, D. und Broek, G.van den (Hrsg.): Tool Integration: Environments and Frameworks. John Wiley & Sons, 1993, S. 1-95.

[Sche98] Scheer, A.-W.: *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen.* Springer, Berlin, 1998.

[Schi92a] Schill, A.: *Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - ein Überblick, Teil 1: Grundlagen.* Informatik Spektrum, 15(2): S. 79-87, 1992.

[Schi92b] Schill, A.: *Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - ein Überblick, Teil 2: Erweiterte RPC-Ansätze.* Informatik Spektrum, 15(3): S. 145-155, 1992.

[Schi93] Schill, A.: *DCE - Das OSF Distributed Computing Environment.* Springer-Verlag Berlin Heidelberg, 1993.

[Schm96] Schmidt, H.: *Ein Application Framework für prozeßorientierte Anwendungssysteme.* Softwaretechnik-Trends, 16(4): S. 44-49, Dez. 1996.

[Schm99] Schmidt, D.C.: *Komponentenbasierte Interoperabilität auf Basis des PRIME Prozeßmetamodells.* Diplomarbeit, RWTH Aachen, 1999.

[Schr97] Schreyjak, St.: *Coupling of Workflow and Component-Oriented Systems.* In: Weck, W., Bosch, J. und Szyperski, C. (Hrsg.), Proceedings of the 2nd International Workshop on Component-Oriented Programming (WCOP '97), 1997, S. 364-368.

[Schu99] Schulze, W.: *Workflow-Management für CORBA-basierte Anwendungen.* Springer-Verlag, 1999.

[ScKR96] Schlenoff, C., Knutilla, A. und Ray, S.: *Unified Process Specification Language: Requirements for Modeling Processes.* Tech. Report, NISTIR 5910, National Institute of Standards and Technology, Gaithersburg, MD, 1996.

[SFGJ99] Schäfer, W., Fuggetta, A., Godart, C. und Jahnke, J.: *Architectural Views and Alternatives*. In: Derniame, J.-C., Kaba, B.A. und Wastell, D. (Hrsg.): Software Process: Principles, Methodology, and Technology. Springer-Verlag Berlin-Heidelberg, LNCS 1500, 1999, S. 95-116.

[Sha*95] Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D. und Zelesnik, G.: *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, 21(4): S. 314-335, 1995.

[ShGa96] Shaw, M. und Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996.

[Shne98] Shneiderman, B.: *Designing the User Interface*. Addison-Wesley, 1998.

[ShWa95] Shams-Aliee, F. und Warboys, B.: *Applying Object-Oriented Modelling to Support Process Technology*. In: Proceedings of the 1st World Conference on Design and Process Technology IDPT, 1995, S.109-115.

[Sieg96] Siegel, J.: *CORBA - Fundamentals and Programming*. John Wiley & Sons, 1996.

[Simm91] Simmonds, I.: *Evolving towards Support for Process Related Sub-Environments*. In: Procs. 7th Intl. Software Process Workshop, 1991, S. 124-126.

[Simm93] Simmonds, I.: *Aerospace Systems Software Engineering Environment*. In: Schefström, D. und Broek, G.van den (Hrsg.): Tool Integration: Environments and Frameworks. John Wiley & Sons, 1993, S. 97-205.

[SJHB96] Schuster, H., Jablonski, St., Heinl, P. und Bussler, Ch.: *A General Framework for the Execution of Heterogenous Programs in Workflow Management Systems*. In: Proc. 1st IFCS Conference on Cooperative Information Systems (CoopIS), Brussels, Belgium, 1996, S. 104-113.

[SlBr93] Slooten, K.van und Brinkkemper, S.: *A Method Engineering Approach to Information Systems Development*. In: Prakash, N., Rolland, C. und Pernici, B. (Hrsg.): Information Systems Development Process. Elsevier Science Publishers, Amsterdam, NL, 1993, S. 167-186.

[SlHo96] Slooten, K.van und Hodes, B.: *Characterizing IS Development Projects*. In: Brinkkemper, S., Lyytinen, K. und Welke, R. (Hrsg.): Method Engineering - Principles of Method Construction. IFIP Chapman & Hall, 1996, S. 29-44.

[SoKe95] Soley, R. und Kent, W.: *The OMG Object Model*. In: Kim, W. (Hrsg.): Object-Oriented Concepts, Databases and Applications. ACM Press, New York, 1995, S. 18-41.

[Sol#83] Sol, H.G.: *A Feature Analysis of Information Systems Design Methodologies*. In: Olle, T.W., Sol, H.G. und Tully, C.J. (Hrsg.): Information Systems Design Methodologies. Elsevier Science Publishers, Amsterdam, NL, 1983, S. 1-7.

[Somm92] Sommerville, I.: *Software Engineering*. Addison-Wesley, 1992.

[SoTM88] Sorenson, P.G., Tremblay, J.-P. und McAllister, A.J.: *The MetaView System for Many specification Environments*. IEEE Software, 5(2): S. 30-38, 1988.

[Srin99] Srinivasan, S.: *Design Patterns in Object-Oriented Frameworks*. Computer, S. 24-32, Feb. 1999.

[Stal81] Stallman, R.M.: *Emacs - The Extensible, Customizable, Self-Documenting Display Editor*. SIGPLAN SIGOA Symposium on Text Manipulation, Special Issue of SIGPLAN Notices, 16(6): S. 147-156, 1981.

[Such87] Suchman, L.: *Plans and Situated Actions: The Problem of Human Machine Communication*. Press Syndicate of the University of Cambridge, 1987.

[SuKN96] Sullivan, K., Kalet, I. und Notkin, D.: *Evaluating the Mediator Method: Prism as a Case Study*. IEEE Transactions on Software Engineering, 22(8): S. 563-579, Aug. 1996.

[SuNo92] Sullivan, K. und Notkin, D.: *Reconciling Environment Integration and Component Independence*. ACM Transactions of Software Engineering and Methodology, 1(3): S. 229-268, July 1992.

[Suns93] Sunsoft: *ToolTalk and Open Protocols: Inter-Application Communication*. SunSoft Press/Prentice Hall, Englewood Cliffs, NJ, 1993.

[Szyp98] Szyperski, C.: *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Tann94] Tannenbaum, A.: *Implementing a Corporate Repository*. John Wiley & Sons, New York, 1994.

[TeRe88] Teitelbaum, T. und Reps, T.: *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*. ACM SIGPLAN Notices, 14(10): S. 75-95, 1988.

[ThMa97] Thomson, H.E. und Mayhew, P.: *Approaches for Software Process Improvement*. Software Process - Improvement and Practice, 3(): S. 3-17, Mar. 1997.

[ThNe92] Thomas, I. und Nejmeh, B.: *Definitions of Tool Integration for Environments*. IEEE Software, 9(2): S. 29-35, 1992.

[Tho*97] Thompson, D., Watkins, D., Exton, W., Garrett, L. und Sajeev, A.S.M.: *Distributed Component Object Model*. In: Krämer, B., Papazoglou, M. und Schmidt, H.-W. (Hrsg.): Information Systems Interoperability. Research Studies Press, Tauton, Somerset, England, 1997, S. 39-75.

[ThTh97] Thayer, R.H. und Thayer, M.C.: *Software Engineering Project Management Glossary*. In: Thayer, R.H. (Hrsg.): Software Engineering - Project Management. IEEE Computer Society Press, 1997, S. 506-529.

[Tolv98] Tolvanen, J.-P.: *Incremental Method Engineering with Modeling Tools*. Dissertation, University of Jyväskylä, 1998.

[Tull91] Tully, C.J.: *Software Process Issues in Software Engineering Environments*. In: Long, F. (Hrsg.), Software Engineering Environments, Ellis Horwood, 1991, S. 1-19.

[VaKa96] Valetto, G. und Kaiser, G.E.: *Enveloping Sophisticated Tools into Process-Centered Environments*. Journal of Automated Software Engineering, 3: S. 309-345, 1996.

[VDI#90] VDI: *VDI-Richtlinie 5005, Software-Ergonomie in der Bürokommunikation*. 1990.

[VeMü97] Verlage, M. und Münch, J.: *Formalizing Software Engineering Standards*. In: Proc. 3rd Intern. Symposium and Forum on Software Engineering Standards (ISESS '97), Walnut Creek, CA, USA, 1997, S. 196-206.

[Vest93] Vestal, S.: *A Cursory Overview and Comparison of Four Architecture Description Languages*. Tech. Report, Honeywell Technology Center, 1993.

[Vest96] Vestal, S.: *MetaH Programmer's Manual*. Tech. Report, Honeywell Technology Center, 1996.

[W3C#98] W3C (World Wide Web Consortium), *Document Object Model (DOM) Level 1 Specification*, W3C Recommendation, REC-DOM-Level-1-19981001, 1998.

[WaJo93] Wakeman, L. und Jowett, J.: *PCTE: The Standard for Open Repositories*. Prentice Hall, Englewood Cliffs, 1993.

[WALM99] Wastell, D., Arbaoui, S., Lonchamp, J. und Montangero, C.: *The Human Dimension of the Software Process*. In: Software Process: Principles, Methodology, and Technology. Springer-Verlag, Berlin Heidelberg, LNCS 1500, 1999, S. 165-199.

[Wand93] Wandmacher, J.: *Software-Ergonomie*. de Gruyter Verlag, Berlin-New York, 1993.

[Warb90] Warboys, B.: *The IPSE 2.5 Project: Process Modelling as a Basis for a Support Environment*. In: Proceedings of the 1st Internatational Conference on System Develeopment Environment and Factories, 1990, S. 77-87.

[Wass90] Wasserman, A.: *Tool Integration in Software Engineering Environments*. In: Proc. Intl. Workshop on Software Engineering Environments, Berlin, Germany, 1990, S. 137-149.

[WaZZ93] Wang, X., Zhao, H. und Zhu, J.: *GRPC: A Communication Cooperation Mechanism in Distributed Systems*. ACM Operating System Review, 27(3): S. 75-86, 1993.

[WeBa99] Weidenhaupt, K. und Bayer, B.: *Prozeßintegrierte Designwerkzeuge für die Verfahrenstechnik*. In: Proc. der Jahrestagung der Gesellschaft für Informatik, Informatik '99, Paderborn, Germany, Oct. 1999, S. 305-313.

[Weid95] Weidenhaupt, K.: *Adaptabilität von Entwicklungsumgebungen: Modellierung und Programmierung*. Diplomarbeit, RWTH Aachen, 1995.

[WFMC96] WFMC: *Workflow Standard - Interoperability Abstract Specification*. Workflow Management Coalition, WFMC-TC-1012, 1996.

[WFMC98a] WFMC: *Workflow Management Application Programming Interface (Interface 2 & 3) Specification*. Workflow Management Coalition, WFMC-TC-1009, Version 2.0, July-98, 1998.

[WFMC98b] WFMC: *WfMC Interface 1: Process Definition Interchange Process Model.* Workflow Management Coalition, WFMC-TC-1016-P, 1998.

[Wije91] Wijers, G.: *Modelling Support in Information Systems Development.* Dissertation, Delft University of Technology, 1991.

[WPJH98] Weidenhaupt, K., Haumer, P., Pohl, K. und Jarke, M.: *Scenarios in System Development: Current Practice.* IEEE Software, 15(2): S. 34-45, 3 1998.

[XMI#99] XMI Partners: *XML Metadata Interchange (XMI) 1.1 RTF Final Report.* Tech. Report, Object Management Group, OMG Document ad/99-10-04, October 20, 1999. (http://www.omg.org./cgi-bin/doc?ad/99-10-04)

[YoCo79] Yourdon, E. und Constantine, L.L.: *Structured Design*: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, 1979.

[Your89] Yourdon, E.: *Modern Structured Analysis.* Prentice Hall, Englewood Cliffs, 1989.

Lebenslauf

Klaus Lambert Weidenhaupt

Geboren am 21.02.1968 in Immerath, jetzt Erkelenz

1974 – 1976	Besuch der Grundschule in Brachelen
1976 – 1978	Besuch der Grundschule in Erkelenz
1978 – 1987	Besuch des Cusanus-Gymnasiums in Erkelenz
	Abitur am 27.06.1987
1987 – 1988	Wehrdienst in Essen und Geilenkirchen
Okt. 1988 – Okt. 1989	Studium der Chemie an der RWTH Aachen
Okt. 1989 – April 1995	Studium der Informatik an der RWTH Aachen
	Diplom am 12.04.1995
Mai 1995 – Juni 2000	Wissenschaftlicher Angestellter am Lehrstuhl für
	Informatik V der RWTH Aachen (Prof. Jarke)
seit Dez. 2000	Wissenschaftlicher Angestellter am
	Philips Forschungslaboratorium Aachen

Aachen, 23.05.2002